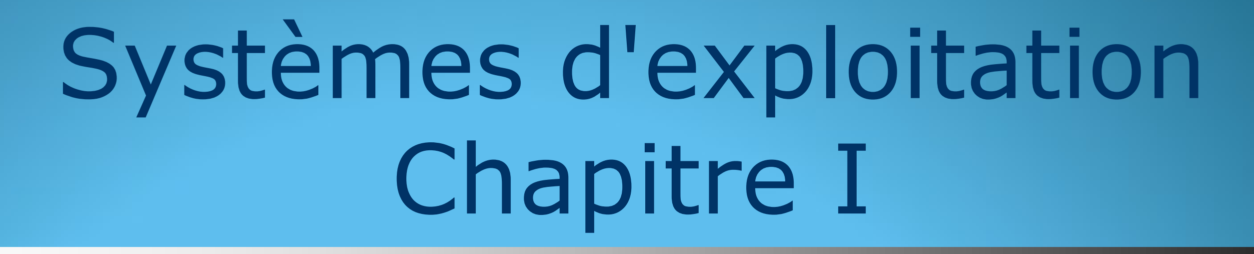


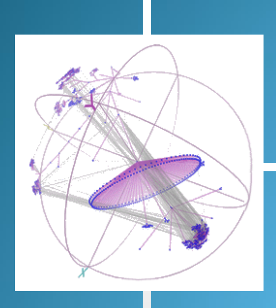
Systèmes d'exploitation
Chapitre I



Introduction
Historique et évolution des ordinateurs

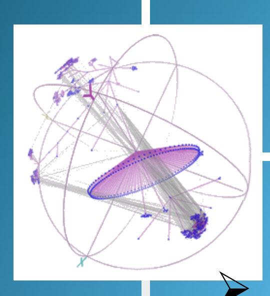
Pr. Omar Megzari
Département d’Informatique
Faculté des Sciences de Rabat

megzari@fsr.ac.ma



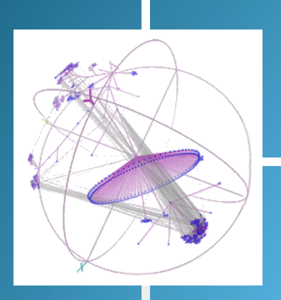
Objectifs du cours

- La partie système s'intéresse aux systèmes d'exploitation modernes et plus particulièrement à :
  - Définitions générales (architectures et buts)
  - Définitions et Historique
  - Gestion des processus
  - Gestion de la mémoire
  - Systèmes de fichiers



Définitions

- INFORmation   autoMATIQUE
  - Science du traitement automatique de l'information
  - Ensemble des applications de cette science, mettant en œuvre des matériels (ordinateurs) et des logiciels
- Système Informatique
  - matériel (hardware) + du logiciel (software)



Définitions

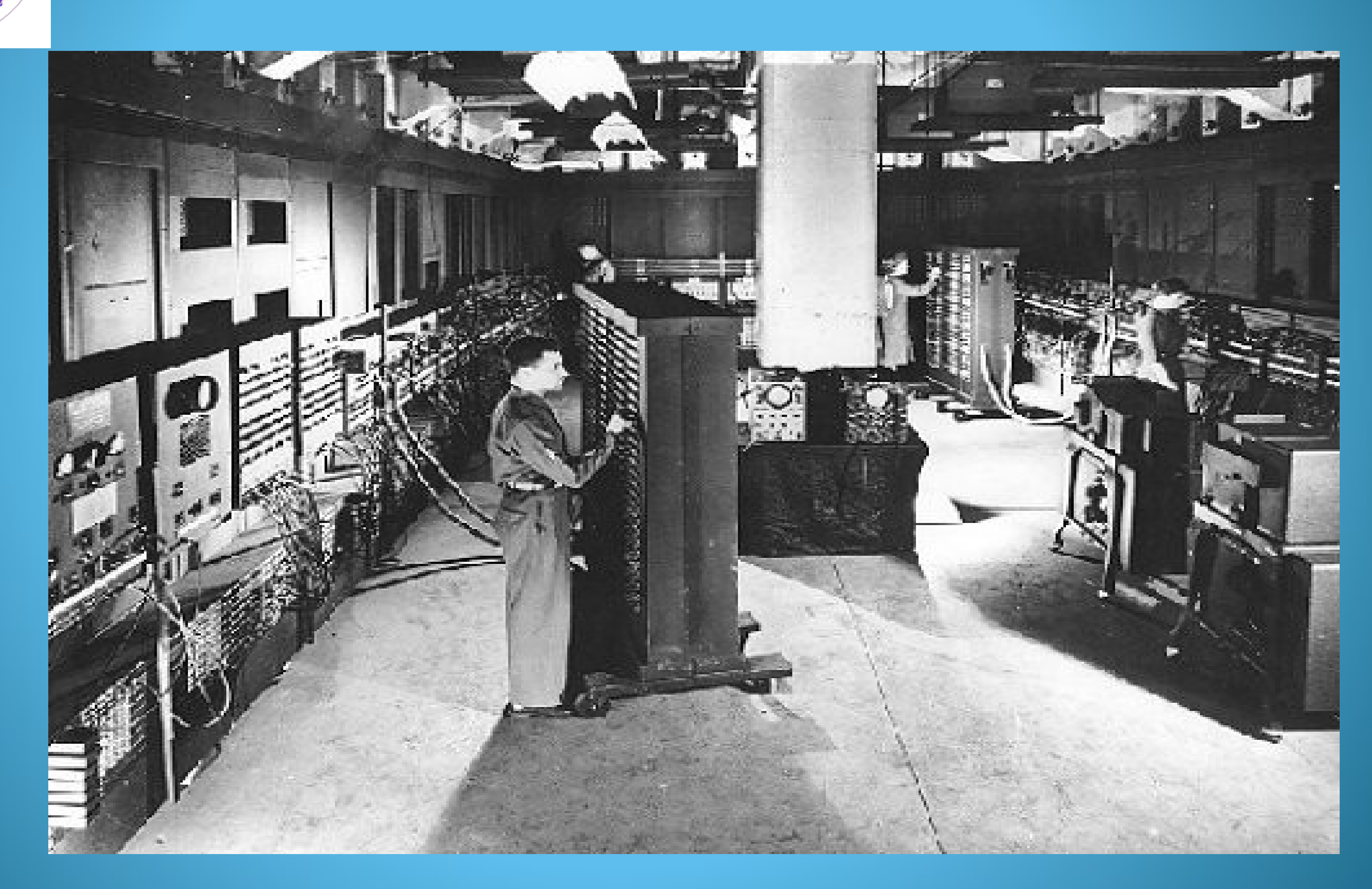
- Ordinateur : « machine à calculer » (calculateur) électronique dotée de mémoires, de moyens de traitement des informations, capable de résoudre des problèmes grâce à l'exploitation automatique de programmes enregistrés
- Programme : ensemble séquentiel d'instructions rédigées pour que l'ordinateur puisse résoudre un problème donné
- Logiciel : ensemble de programmes relatif à des traitements d'informations (ex. Windows, Word...)

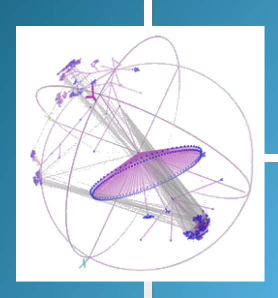


Ordinateur et changements technologiques

- Première génération: Tubes électroniques (lampes à vide)
- Deuxième génération: transistors
- Troisième génération: circuits intégrés
- Quatrième génération: microprocesseurs.
- Cinquième génération: intelligence artificielle.

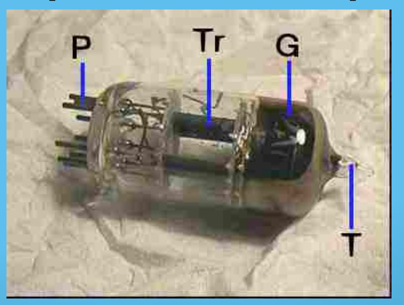
Disposé en une sorte de U de 6 mètres de largeur par 12 mètres de longueur et pesait 30 tonnes.





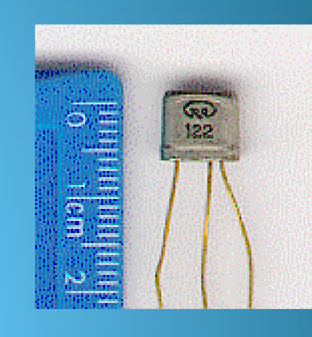
Première génération 1949-1957

- Ordinateur à cartes perforées et à bandes magnétiques
- Programmation physique en langage machine
- Calcul numérique (trigonométrie)
- Appareils immenses, lourds, énergie élevée
- Utilisation de tubes à vide
- Prix élevé / capacité et performance.

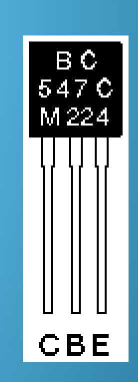


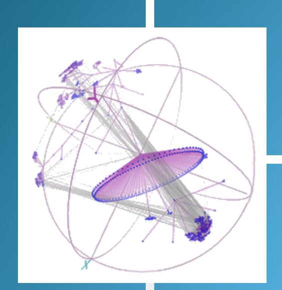


Deuxième génération 1958 - 1964



- Utilisation de transistors
- Transistor => augmentation de la fiabilité
- Utilisation de mémoires de masse pour le stockage périphériques.
- Temps d’accès moyen (de l’ordre de la micro-seconde).
- Fonctionnement séquentiel des systèmes de programmation (langages évolués):FORTRAN
- Mainframes





Troisième génération 1965-1971

- Miniaturisation des composants (circuits intégrés)
- Apparition des systèmes d'exploitation
- Concepts de temps partagés
- Machines polyvalentes et de capacité variée
- Appareils modulaires et extensibles
- Multitraitement (plusieurs programmes à la fois)
- Télétraitement (accès par téléphone)
- UNIX
- Mini ordinateurs



Quatrième génération 1971-1982

- Miniaturisation extrêmes des composants
- Apparition des microprocesseurs
- Diversification des champs d’application
- Apparition de la micro-informatique
- L’aspect logiciel prend le pas sur l’aspect matériel

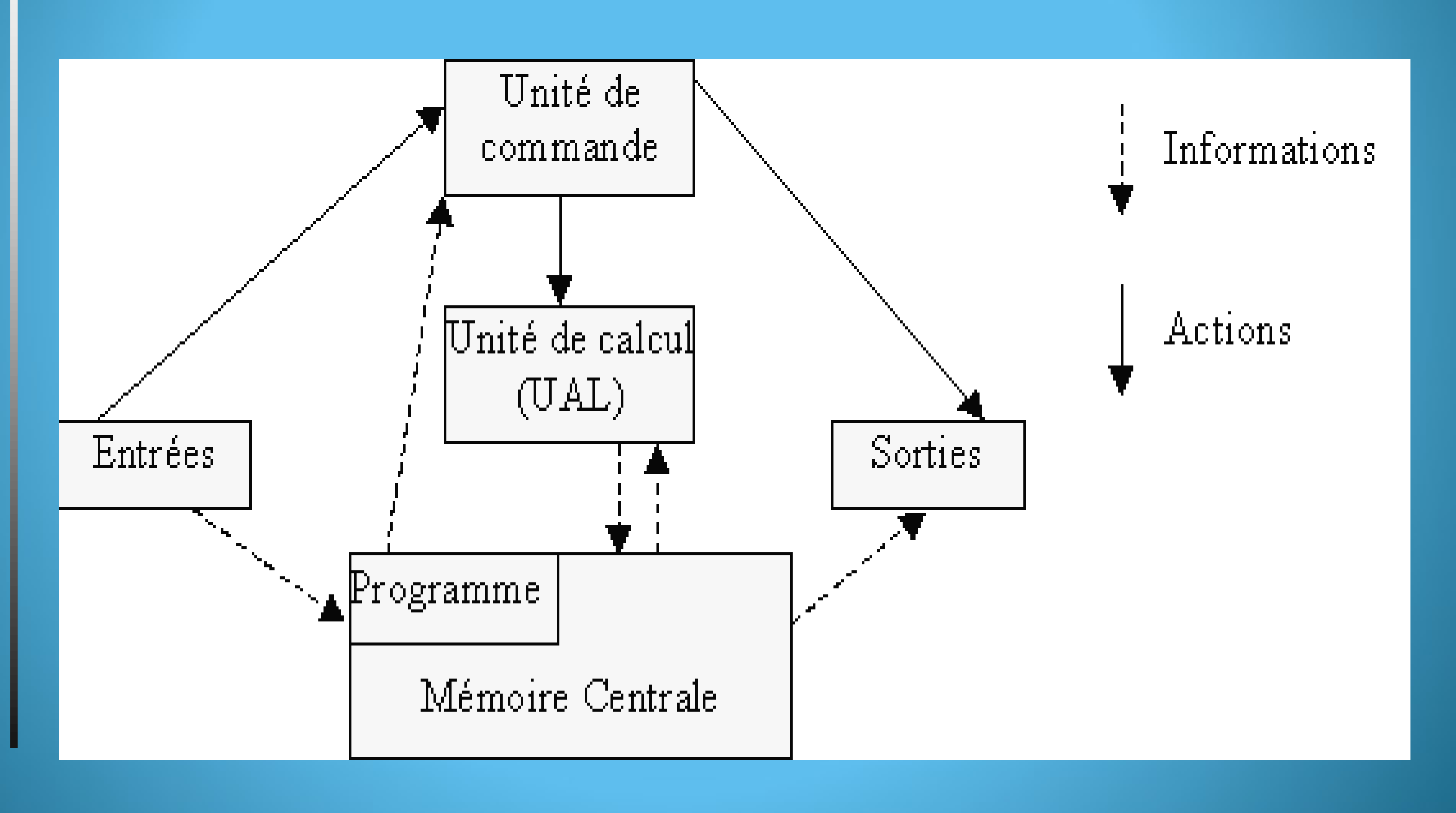
Cinquième génération

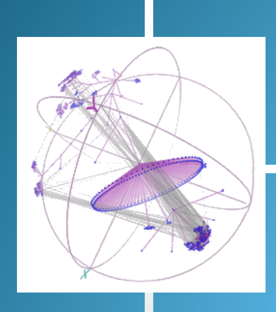


- Miniaturisation des composants poussée à l'extrême
- Vitesse proche de celle de la lumière.
- Processeurs en parallèle
- Nouvelles structures et représentations des données.

Schéma de la machine de Von Newman

UAL = unité arithmétique et logique

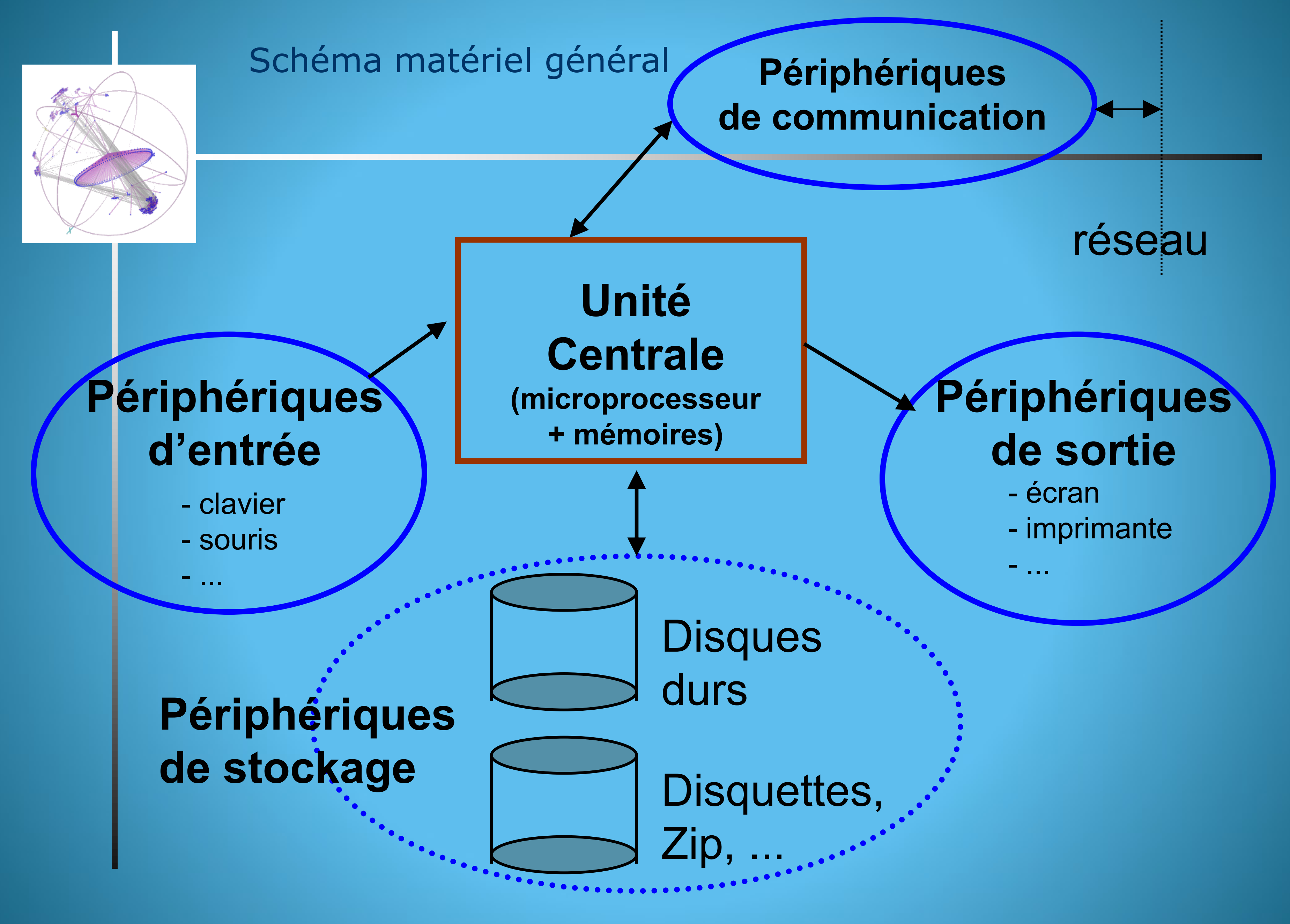




Machine de Von Newman

Ces dispositifs permettent la mise en œuvre des fonctions de base d'un ordinateur :

-le stockage de données,
-le traitement des données,
-le mouvement des données et
-le contrôle des périphériques.


Schéma matériel général
Périphériques
de communication
réseau
Unité
Centrale
(microprocesseur
+ mémoires)
Périphériques
d'entrée
- clavier
- souris
- ...
Périphériques
de sortie
- écran
- imprimante
- ...
Disques
durs
Disquettes,
Zip, ...
Périphériques
de stockage



L’unité centrale

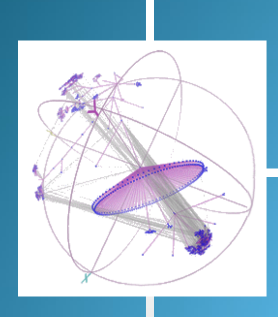
Le (micro)processeur ou CPU :

Central Processing Unit

Il exécute les programmes :

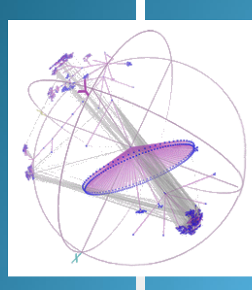
un programme est une suite d’instructions

- Unité arithmétique et logique (UAL) et Unité de commande

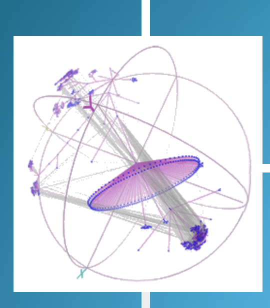


Mémoire vive : RAM

- RAM (Random Access Memory)
  - Permet de stocker des informations lorsqu’elle est alimentée électriquement
  - Lecture / Écriture
  - Mémoire volatile : contient des programmes et des données en cours d’utilisation
  - Capacité variable selon les ordinateurs
    - > à 4 go sur les PCs



Barrette de mémoire RAM

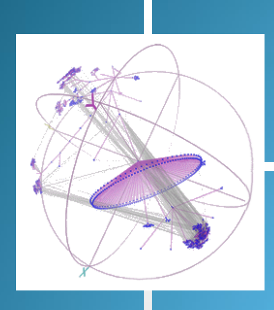


Mémoire vive : RAM



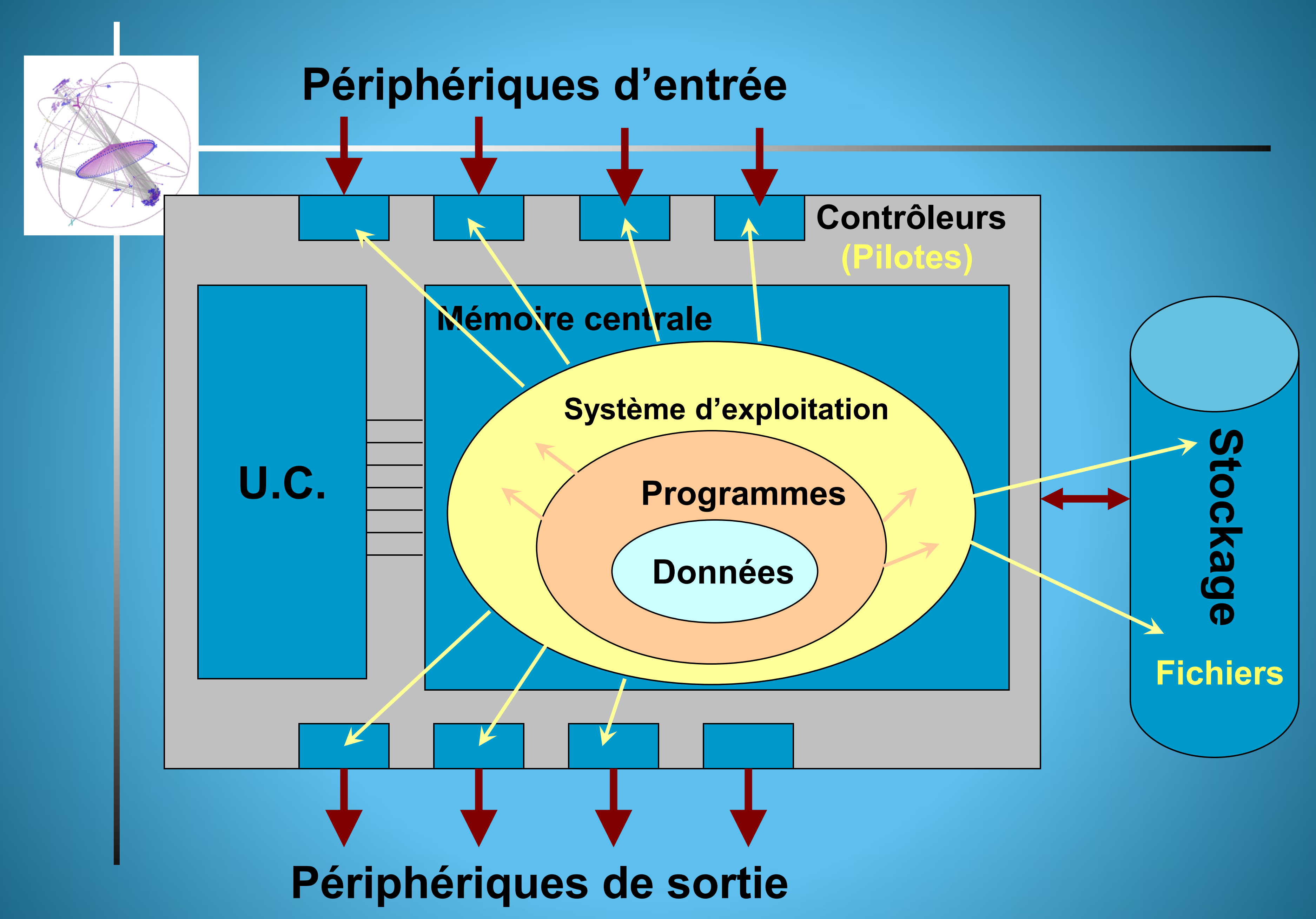
Mémoire morte : ROM

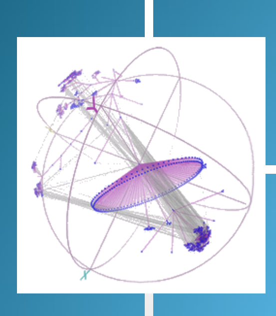
- ROM (Read Only Memory)
  - En lecture seule
  - Mémoire permanente
  - Contient les programmes de base au démarrage de l’ordinateur (initialisation de l’ordinateur, initialisation de périphériques, lancement du système d’exploitation…BIOS)



Les périphériques

- Les périphériques de stockage
- Les périphériques d’entrée
- Les périphériques de sortie
- Les périphériques de communication


Périphériques d'entrée
Contrôleurs
(Pilotes)
Mémoire centrale
U.C.
Système d'exploitation
Programmes
Données
Stockage
Fichiers
Périphériques de sortie



Périphériques d’entrée

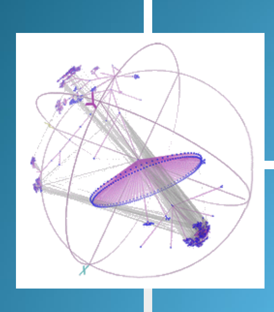


- Permettent d’envoyer des informations à l’Unité Centrale



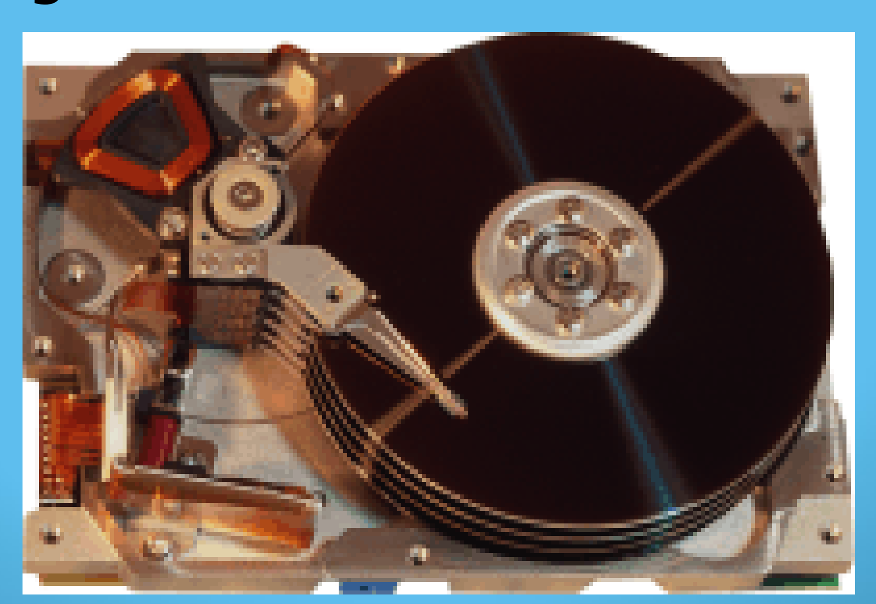
Périphériques de sortie

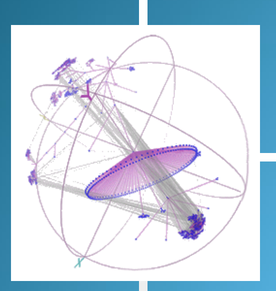
- Permettent d ’envoyer les résultats à l’extérieur de l’Unité Centrale
  - Écrans
    - taille (en pouce), résolution...
  - Imprimantes
    - matricielles, jet d ’encre, laser
  - Enceintes



Les périphériques de stockage

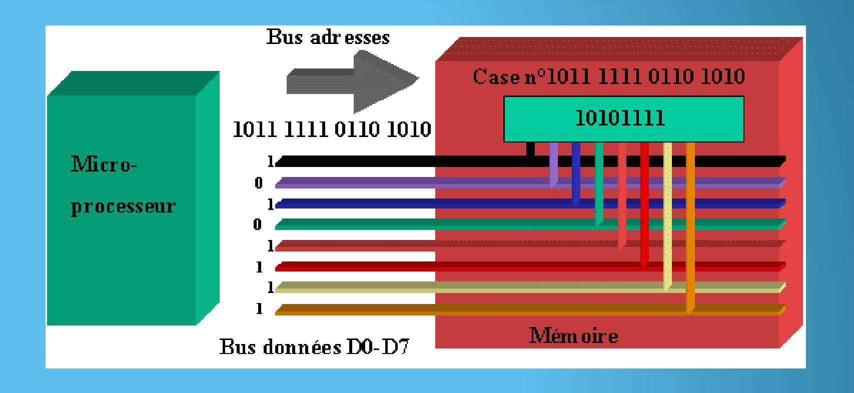
- CD-ROM (650 Mo et 800 Mo)
- DVD (4,7 à 17 Go)
- Disque dur > 320 Go
- Différence entre RAM et supports de stockage

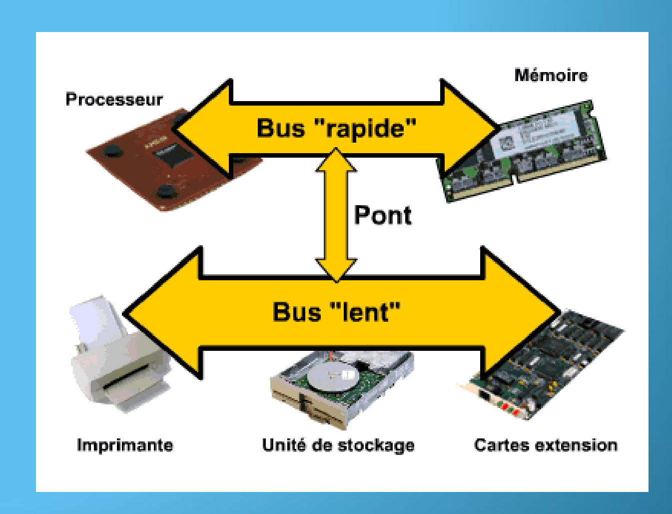


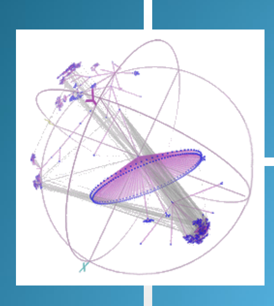


Les BUS

- Permettent le transfert des données entre les composants de l'ordinateur
- Différentes technologies ➔ plus ou moins grande capacité de transfert





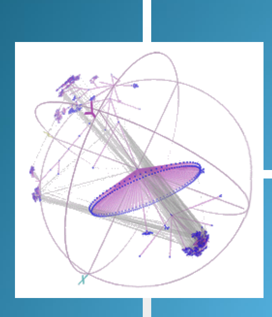


Systèmes d'Exploitations

Qu'est-ce qu'un SE ?

Deux visions :

- Une ***interface*** entre l'utilisateur et le matériel.
  - Cacher les spécificités matérielles à l'utilisateur.
- Un ***gestionnaire de ressources :*** un programme qui gère les ressources de l'ordinateur (processeur, mémoire, périphériques, etc.).
  - Savoir quelles ressources sont disponibles
  - Savoir qui utilise quoi, quand, combien, etc.
  - Allouer/Libérer les ressources efficacement.

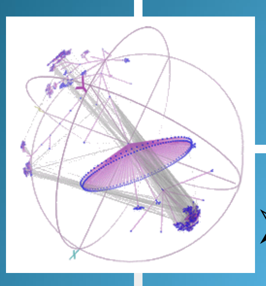


Plus formellement

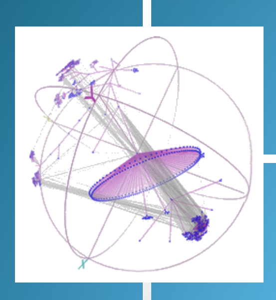
Plus formellement, c'est un ensemble de programmes dont la fonction est de :

- Gérer les ressources
  - Physiques : processeur, mémoire, disques, etc.
  - Logiques : fichiers et bases de données etc.
- Contrôler les entrées-sorties
- ordonnancer les travaux
- gérer les erreurs
- fournir des mécanismes de sécurité

En conclusion : servir d'interface entre l'utilisateur et la machine.



- Le SE fonctionne exactement comme un programme ordinaire :
  - Il est exécuté par le processeur de la même manière.
  - La différence principale est sa fonction : il dirige le processeur sur l'utilisation des ressources et la manière d'exécuter les autres programmes.
- Une partie, le ***noyau*** (kernel)
  - reste en mémoire
  - contient les fonctions les plus utilisées du SE
    - gère les processus, leur ordonnancement,
    - les communications interprocessus,
    - la mémoire et
    - les accès aux ressources



Architecture générale

utilisateurs

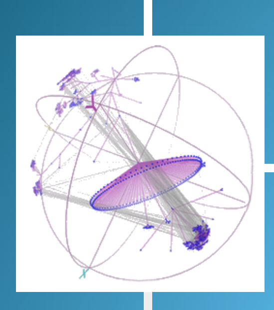
programmes d'applications	programmes utilitaires	ateliers de programmation

éditeur de texte	assembleur	compilateur	éditeur de liens	chargeur	metteur au point

système d'exploitation

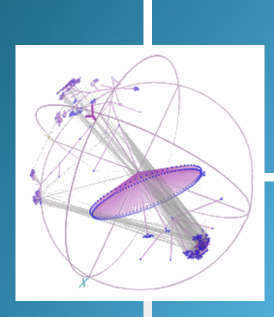
gestion processeur	gestion mémoire	gestion données	gestion périphériques	gestion communication

machine nue



Les différents composants

- Gestion :
  - des processus
  - de la mémoire
  - des périphériques de stockage auxiliaires
  - des Entrées Sorties (E/S)
  - des fichiers
  - …
- Protection de l’intégrité
- Réseau
- Interpréteur de commandes

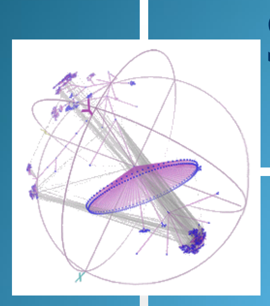


Développement des SE

- La théorie des SE a été développée surtout dans les années 1960 (!!)
- A cette époque, il y avait des machines très peu puissantes avec lesquelles on cherchait à faire des applications comparables à celles d'aujourd'hui
- Ces machines devaient parfois desservir des dizaines d'usagers!
- Dont le besoin de développer des principes pour optimiser l'utilisation d'un ordinateur.
- Principes qui sont encore utilisés

Évolution historique des SE

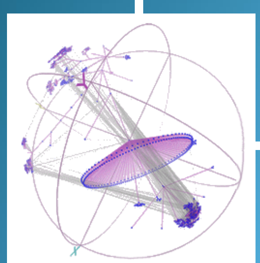
- Le début: routines d’E/S, amorçage système
- Systèmes par lots simples
- Systèmes par lots multiprogrammés
- Systèmes à partage de temps
- Ordinateurs personnels
- SE en réseau
- SE répartis
- *Les problèmes et solutions qui sont utilisés dans les systèmes simples se retrouvent souvent dans les systèmes complexes.*



Systèmes de traitement par lots *(batch)* simples

- Sont les premiers SE (mi-50)
- L'usager soumet un job (ex: sur cartes perforées) à un opérateur
- L'opérateur place un lot de plusieurs jobs sur le dispositif de lecture
- Un programme, le moniteur, gère l'exécution de chaque programme du lot
- Le moniteur est toujours en mémoire et prêt à être exécuté
- Un seul programme à la fois en mémoire, et les programmes sont exécutés en séquence
- La sortie est normalement sur un fichier, imprimante ou ruban magnétique…

Un ordinateur principal
(mainframe) du milieu des années ‘60





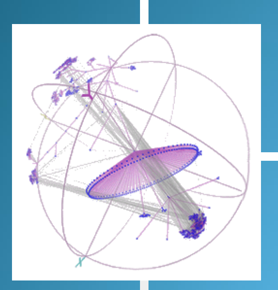
disques

UCT
(mémoire probablem.
autour de 500K)

rubans

lecteur de cartes

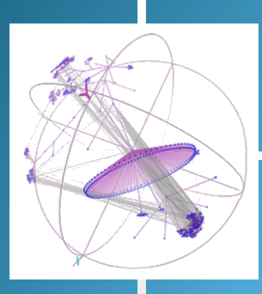
console opérateur



Cartes perforées…

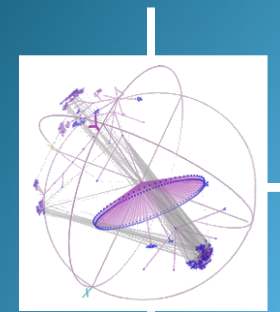
0123456789 ABCDEFGHI JKLMNOPQR STUVWXYZ & . < - $ * / , % # @
UNIVERSITY OF FLORIDA
AGRIGATOR
ICON © 1997

Une ligne de données ou de programme était codée dans des trous qui pouvaient être lus par la machine

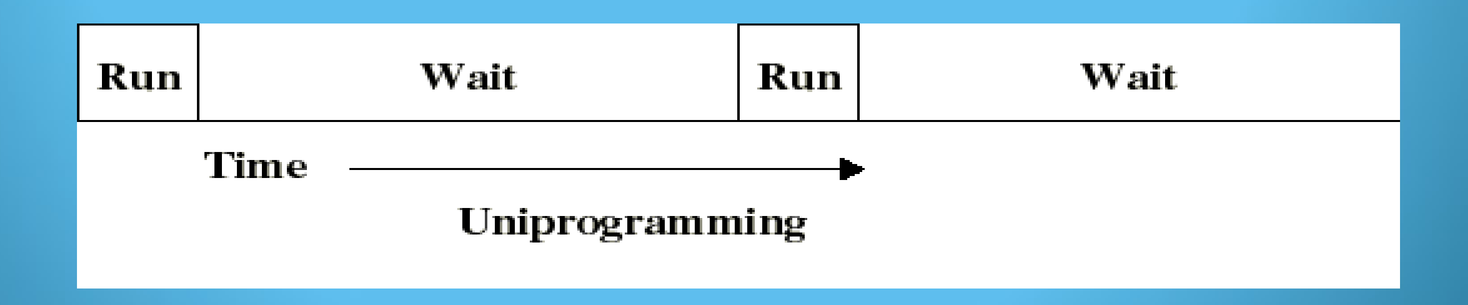


Lecteur de cartes perforées

Traitement par lots multiprogrammé



- Les opérations E/S sont extrêmement lentes (comparé aux autres instructions)
- Même avec peu d'E/S, un programme passe la majorité de son temps à attendre
- Donc: pauvre utilisation de l'UCT lorsqu'un seul programme usager se trouve en mémoire





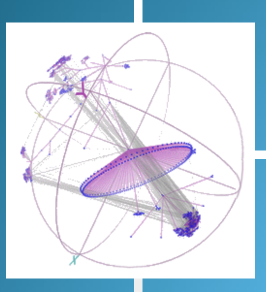
Traitement par lots multiprogrammé

- Si la mémoire peut contenir plusieurs programmes, l’UCT peut exécuter un autre programme lorsqu’un programme attend une E/S
- C’est de la multiprogrammation

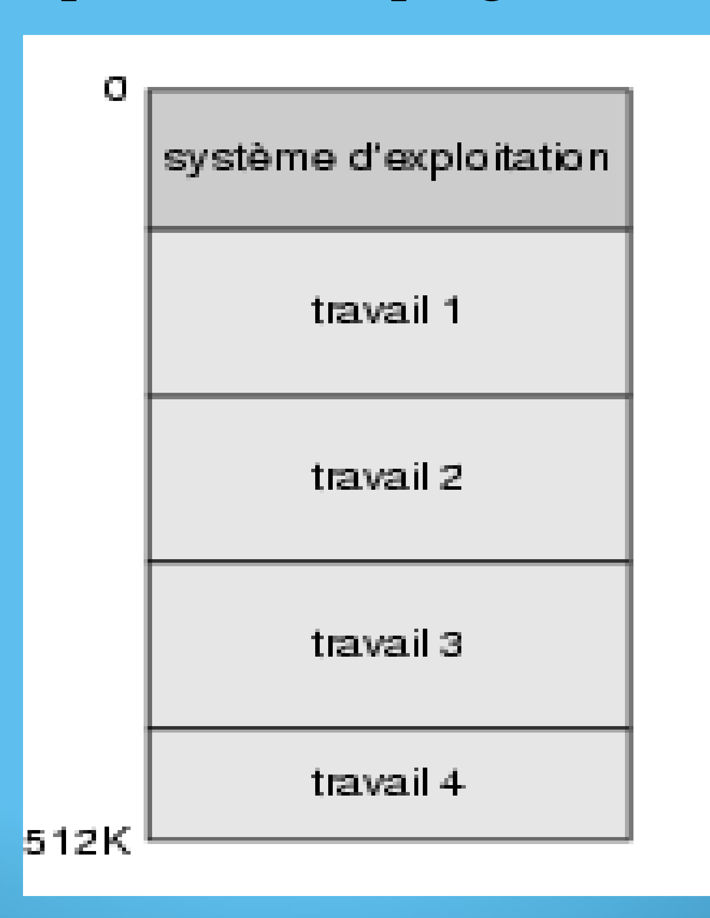
Program A	Run	Wait			Run	Wait			
Program B	Wait	Run	Wait			Run	Wait		
Program C	Wait		Run	Wait			Run	Wait	
Combined	Run A	Run B	Run C	Wait	Run A	Run B	Run C	Wait	

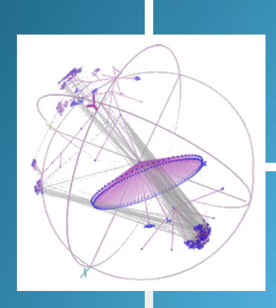
Time →

Multiiprogramming with three programs



Plusieurs programmes en mémoire pour la multiprogrammation





Spoule ou spooling

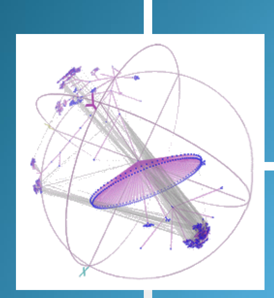
- Au lieu d’exécuter les travaux au fur et à mesure qu’ils sont lus, les stocker à l’avance sur une mémoire secondaire (disque)
- Puis choisir quels programmes exécuter et quand
- La mémoire secondaire contenait aussi les données d’E/S



Systèmes à temps partagé (TSS)

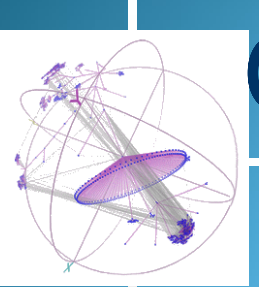
Terminaux
'stupides'

ordinateur principal
(mainframe)



Systèmes à temps partagé (TSS)

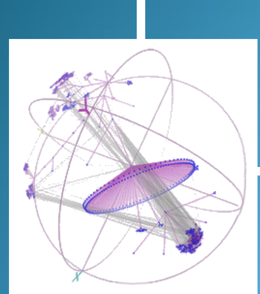
- Le traitement par lots multiprogrammé ne supporte pas l'interaction avec les usagers
  - **excellente utilisation des ressources mais frustration des usagers!**
- TSS permet à la multiprogrammation de desservir plusieurs usagers simultanément
- Le temps d'UCT est partagé par plusieurs usagers
- Les usagers accèdent simultanément et interactivement au système à l'aide de terminaux



Ordinateurs Personnels (PCs)

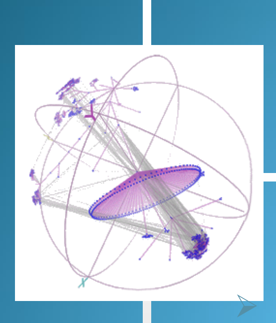
- Au début, les PCs étaient aussi simples que les premiers ordinateurs
- Le besoin de gérer plusieurs applications en même temps conduit à redécouvrir la multiprogrammation
- Le concept de PC isolé évolue maintenant vers le concept d'ordinateur de réseau *(network computer),* donc extension des principes des TSS.

Aujourd'hui



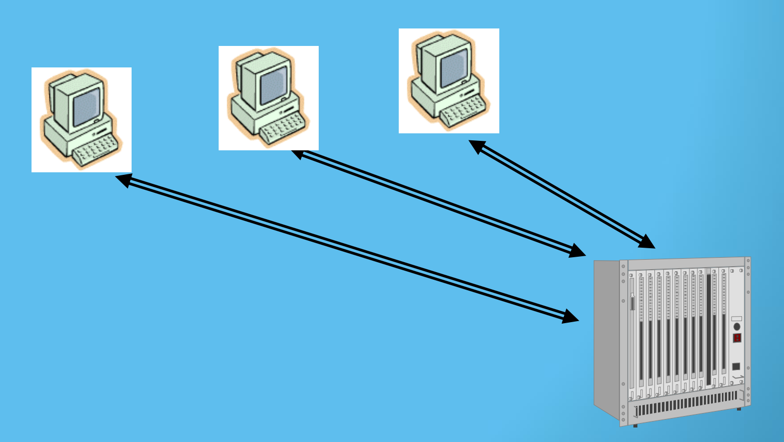
Terminaux
'intelligents' (PCs)'

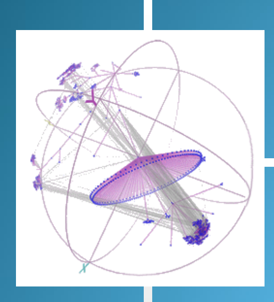
ordinateur principal
(mainframe ou serveur)



Aujourd’hui

- Plusieurs PC (clients) peuvent être desservis par un ordinateur plus puissant (serveur) pour des services qui sont trop complexes pour eux (clients/serveurs, bases de données, etc)





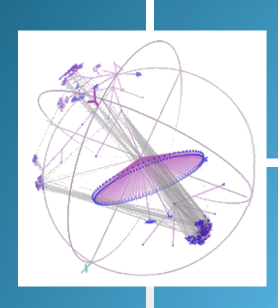
D’autres Systèmes

- Systèmes d’exploitation répartis:
  - Le SE exécuté à travers un ensemble de machines qui sont reliées par un réseau
- Systèmes multiprocesseurs
- Systèmes parallèles
- Systèmes à temps réel



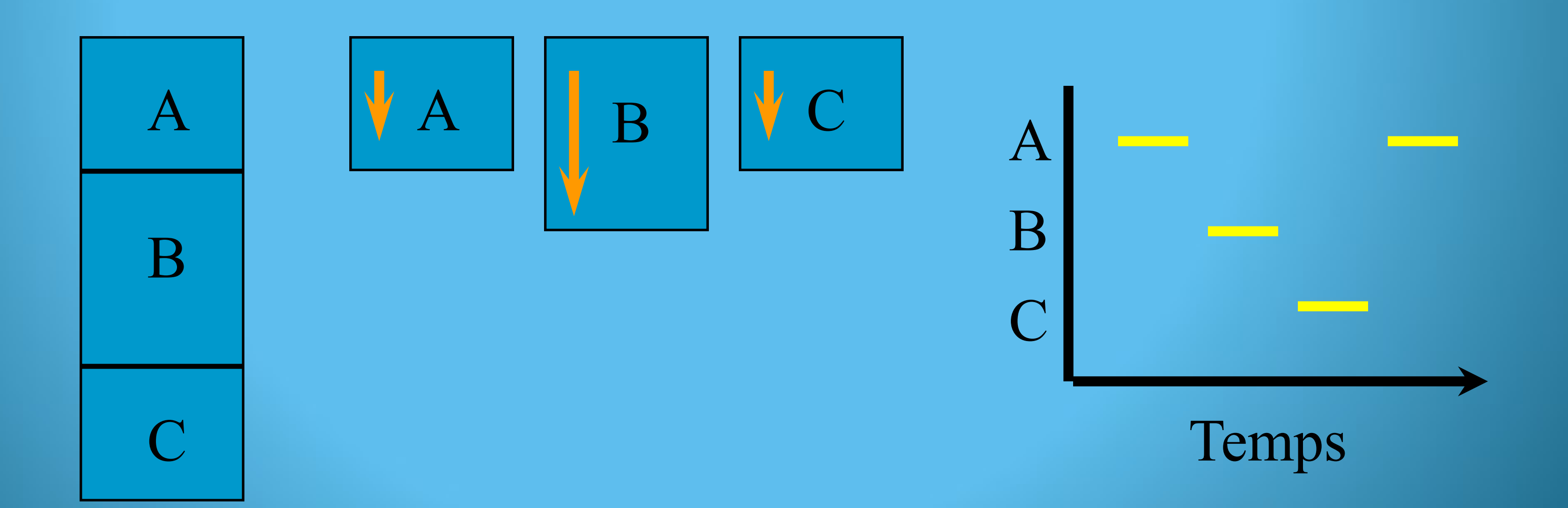
Systèmes d'exploitation
Chapitre II

Gestion des processus



Processus

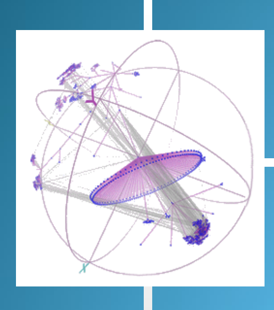
- “Un programme qui s’exécute”
- Les ordinateurs autorisent maintenant plusieurs processus simultanément (pseudo parallélisme)





État de processus

- Au fur et a mesure qu'un processus s'exécute, il change d'état
  - nouveau: le processus vient d'être créé
  - exécutant-running: le processus est en train d'être exécuté par l 'UCT
  - attente-waiting: le processus est en train d'attendre un événement (p.ex. la fin d 'une opération d'E/S)
  - prêt-ready: le processus est en attente d'être exécuté par l 'UCT
  - terminé: fin d'exécution



Etats des processus

- Changement d'état des processus

Terminaison

Nouveau

Actif

élu

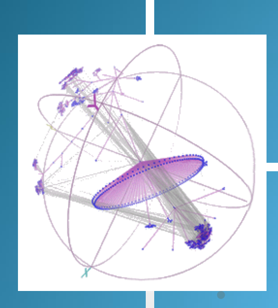
Attente E/S

Interruption

Prêt

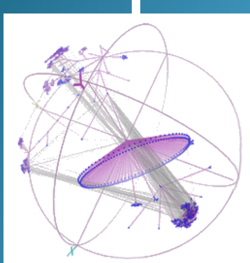
Attente

E/S terminée



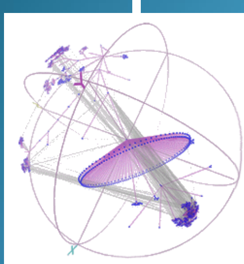
États Nouveau, Terminé

- **Nouveau**
  - Le SE a créé le processus
    - a construit un identificateur pour le processus
    - a construit les tableaux pour gérer le processus(PCB)
  - mais ne s’est pas encore engagé à exécuter le processus (pas encore admis)
    - pas encore alloué des ressources
  - La file des nouveaux travaux est souvent appelée spoule travaux (job spooler)
- **Terminé:**
  - Le processus n’est plus exécutable, mais ses données sont encore requises par le SE (comptabilité, etc.)



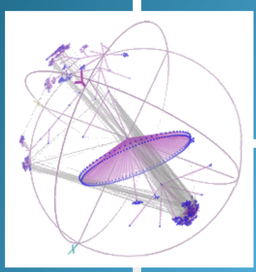
Transitions entre processus

- **Prêt → Exécution**
  - Lorsque **l'ordonnanceur** UCT choisit un processus pour exécution
- **Exécution → Prêt**
  - Résultat d'une interruption causée par un événement indépendant du processus
    - Il faut traiter cette interruption, donc le processus courant perd l'UCT
      - Cas important: le processus à épuisé son intervalle de temps (quantum)



Transitions entre processus

- **Exécution → Attente**
  - Lorsqu’un processus fait requête d’un service du SE que le SE ne peut offrir immédiatement (interruption causée par le processus lui-même)
    - un accès à une ressource pas encore disponible
    - initie une E/S: doit attendre le résultat
    - a besoin de la réponse d’un autre processus
- **Attente → Prêt**
  - lorsque l'événement attendu se produit



Sauvegarde d’informations du processus

- En multiprogrammation, un processus s’exécute sur l’UCT de façon intermittente
- Chaque fois qu’un processus reprend l’UCT (transition prêt $\rightarrow$ exécution) il doit la reprendre dans la même situation où il l’a laissée (même contenu des registres UCT, etc.)
- Donc au moment où un processus sort de l’état exécution il est nécessaire de sauvegarder ses informations essentielles, qu’il faudra récupérer quand il retourne à cet état

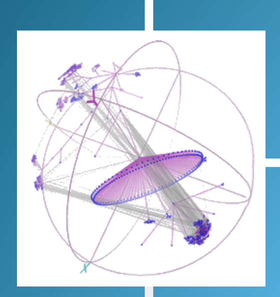


PCB = Process Control Block

Représente la situation actuelle d'un processus, pour le reprendre plus tard

pointeur	état de processus
numéro de processus	
compteur programme	
registres	
limites mémoire	
liste des fichiers ouverts	
. . .	

Registres UCT

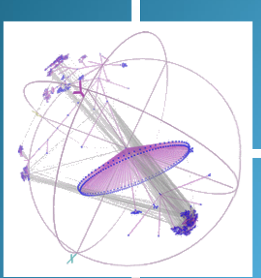


Process Control Block

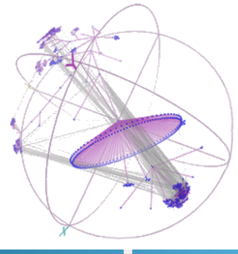
Le PCB contient entre autres:

- pointeur: les PCBs sont rangés dans des listes enchaînées (à voir)
- état de processus: ready, running, waiting…
- compteur programme: le processus doit reprendre à l ’instruction suivante
- autres registres UCT
- bornes de mémoire
- fichiers qu’il a ouvert
- etc.,

Commutation de processus



- appelé aussi commutation de contexte ou context switching
- Quand l'UCT passe de l'exécution d'un processus 0 à l'exécution d'un proc 1, il faut
  - mettre à jour le PCB de **0**
  - sauvegarder le PCB de **0**
  - reprendre le PCB de **1**, qui avait été sauvegardé avant
  - remettre les registres d 'UCT, compteur d 'instructions etc. dans la même situation qui est décrite dans le PCB de **1**



Commutation de processeur (context switching)

processus P_0 | système d'exploitation | processus P_1

interruption ou appel système

exécuter

sauver l'état dans le PCB_0

inact

recharger l'état depuis le PCB_1

inactif

exécuter

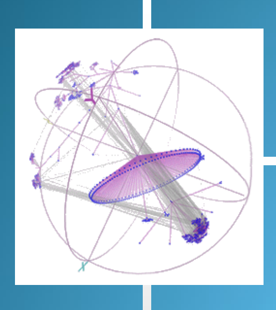
interruption ou appel système

sauver l'état dans le PCB_1

inact

recharger l'état depuis le PCB_0

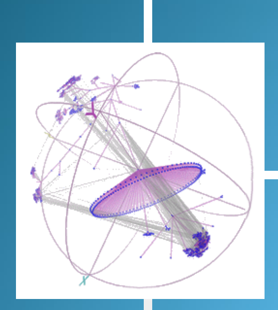
exécuter



La pile d'un processus

Il faut aussi à sauvegarder entre autre: la pile d'un processus

- Quand un processus fait appel à une procédure, à une méthode, etc., il est nécessaire de mettre dans une pile l'adresse à laquelle le processus doit retourner après avoir terminé cette procédure, méthode, etc.
- Aussi on met dans cette pile les variables locales de la procédure qu'on quitte, les paramètres, etc., pour les retrouver au retour
- Donc il y a normalement une pile d'adresses de retour après interruption et une pile d'adresses de retour après appel de procédure
  - Ces deux piles fonctionnent de façon semblable, mais sont normalement séparées
- Les informations relatives à ces piles (base, pointeur…) doivent aussi être sauvegardées au moment de la commutation de contexte



La Pile d’un processus

P

Appel A

A

Appel B

B

Données B

Données A

Données P

PILE



Commutation de processus

Comme l'ordinateur n'a, la plupart du temps, qu'un processeur, il résout ce problème grâce à un pseudo-parallélisme

Figure 1 Le multi-tâche



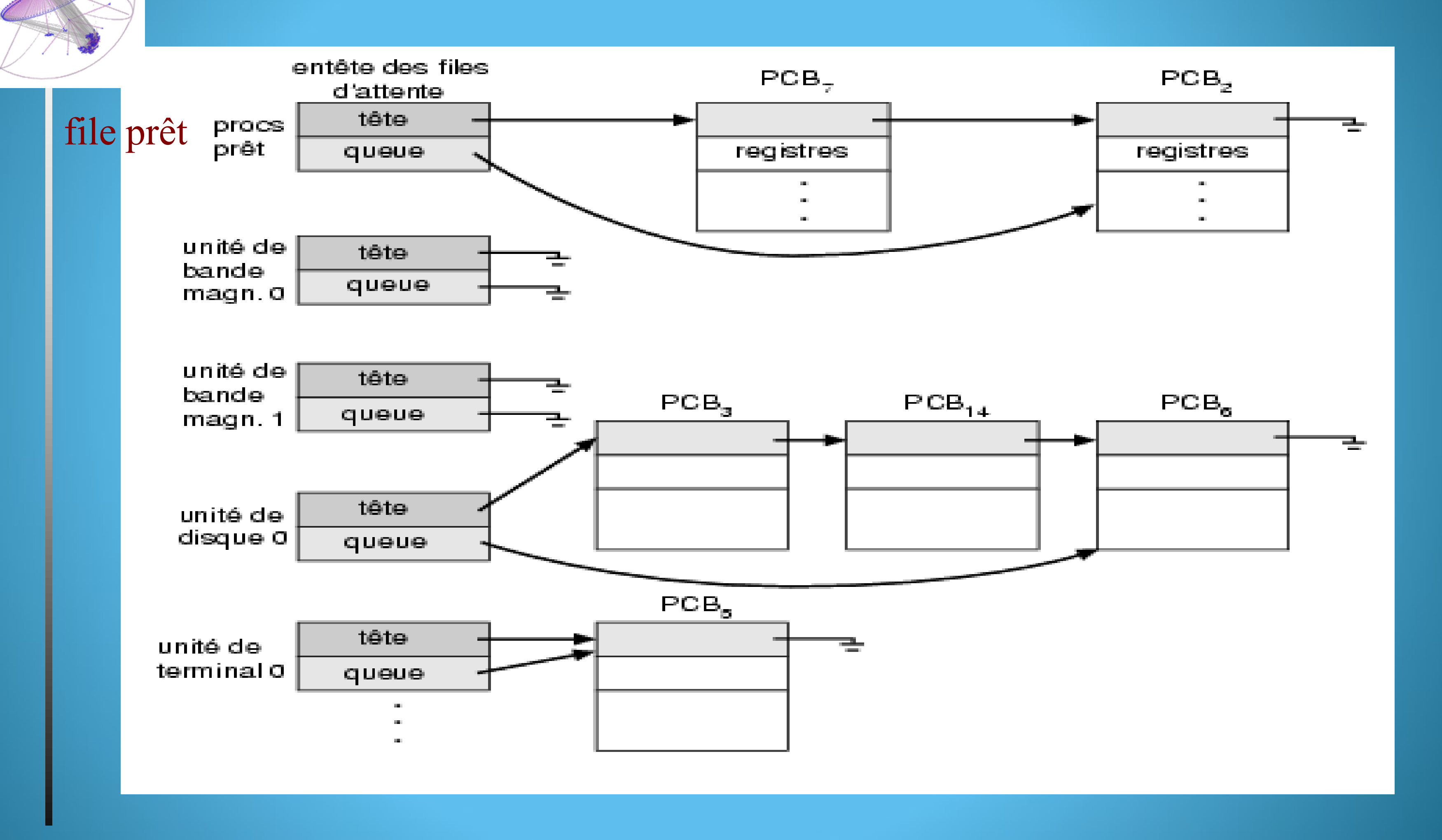
Files d'attente

- Les ressources d'ordinateur sont souvent limitées par rapport aux processus qui en demandent
- Chaque ressource a sa propre file de processus en attente
- En changeant d'état, les processus se déplacent d 'une file à l`autre
  - File prêt: les processus en état prêt=ready
  - Files associés à chaque unité E/S
  - etc.

Files d’attente

Ce sont les PCBs qui sont dans les files d’attente

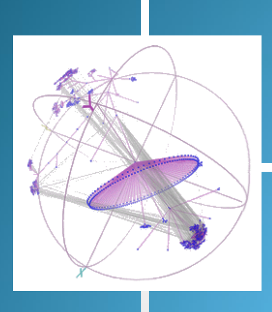
file prêt



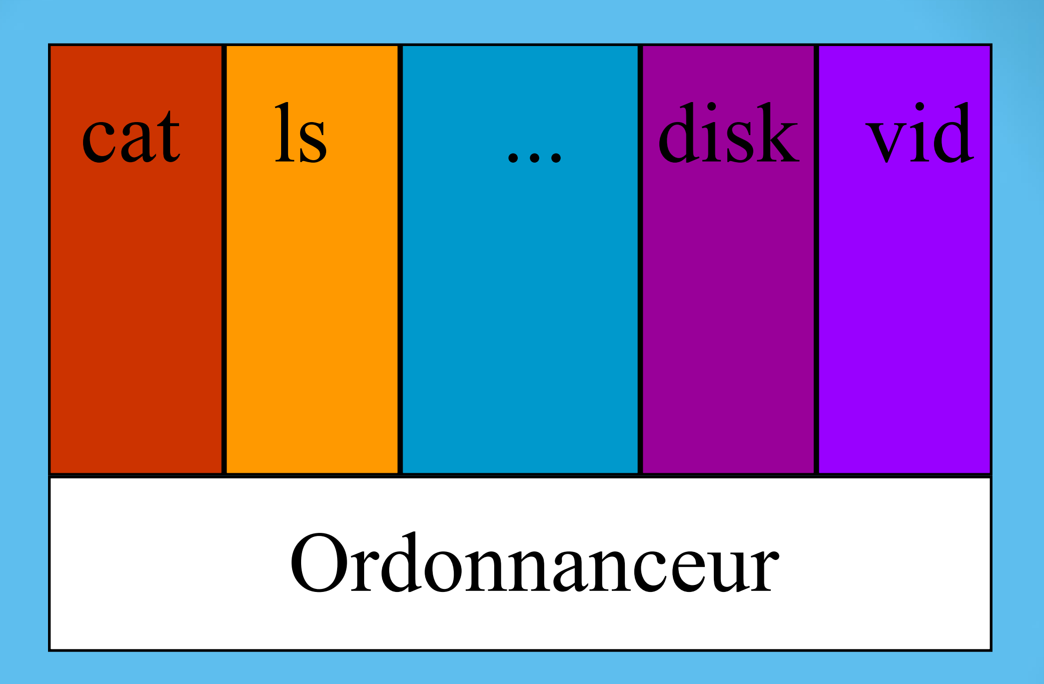


Les PCBs

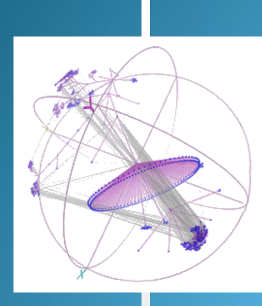
Les PCBs ne sont pas déplacés en mémoire pour être mis dans les différentes files: ce sont les pointeurs qui changent.



Ordonnanceur



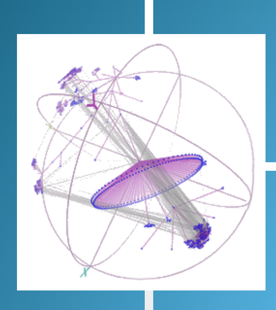
- Tous les services sont des processus



Ordonnanceurs (schedulers)

Programmes qui gèrent l'utilisation de ressources de l'ordinateur

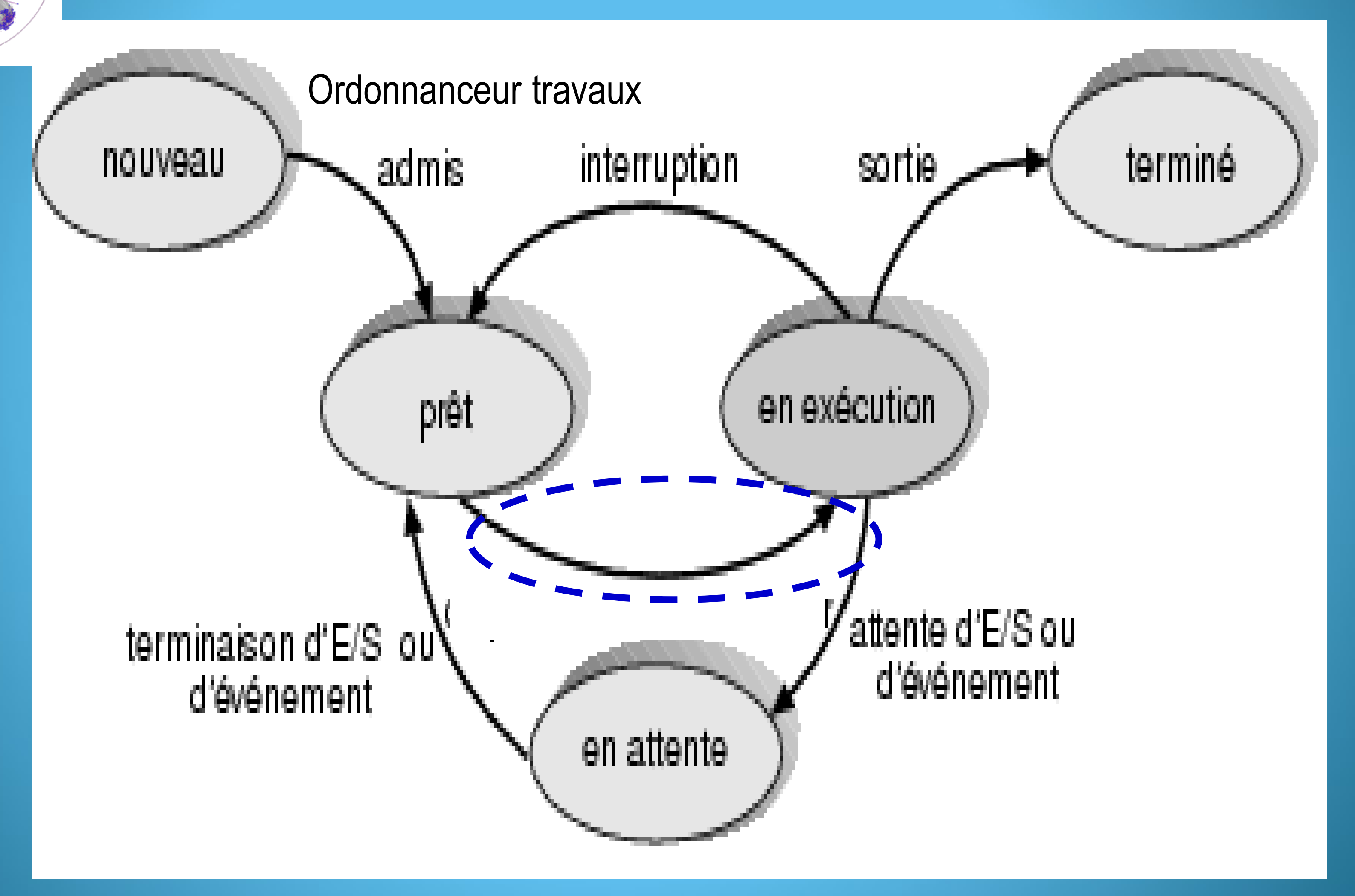
- Trois types d'ordonnanceurs :
  - À court terme = ordonnanceur processus: sélectionne quel processus doit exécuter la transition prêt $\rightarrow$ exécution
  - À long terme = ordonnanceur travaux: sélectionne quels processus peuvent exécuter la transition nouveau $\rightarrow$ prêt (événement *admitted*) (de spoule travaux à file prêt)
  - À moyen terme: répond au manque de mémoire

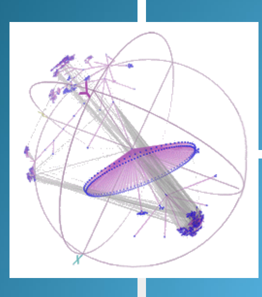


Ordonnanceurs

- L'ordonnanceur à court terme est exécuté très souvent (millisecondes), Il faut donc que ça aille très vite
  - typiquement de 1 à 1000 microsecondes
- L'ordonnanceur à long terme doit être exécuté beaucoup plus rarement: il contrôle le niveau de multiprogrammation
  - doit établir une équilibre entre les travaux liés à l'UCT et ceux liés à l'E/S de sorte à ce que les ressources de l'ordinateur soient bien utilisées

Ordonnanceur travaux = long terme
Ordonnanceur processus = court terme





Ordonnancement de processus (court terme)

file des procs. prêts

UC

E/S

file d'attente d'E/S

requêtes d'E/S

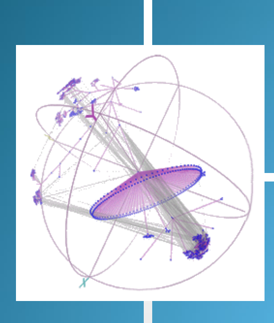
tranche de temps expirée

le fils s'exécute

création d'un fils

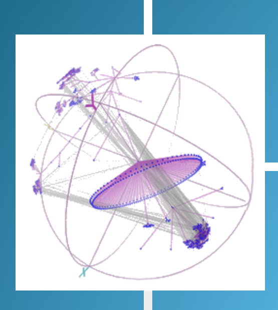
l'interruption se produit

en attente d'une interruption



Ordonnanceur à moyen terme

- Le manque de ressources peut parfois forcer le SE à *suspendre* des processus
  - ils seront plus en concurrence avec les autres pour des ressources
  - ils seront repris plus tard quand les ressources deviendront disponibles
- Ces processus sont enlevés de mémoire centrale et mis en mémoire secondaire, pour être repris plus tard
  - ‘swap out’, ‘swap in’ , va-et-vient



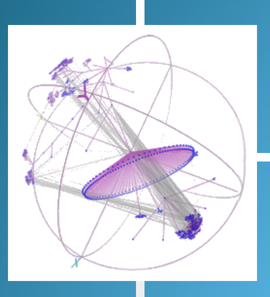
Concepts de base

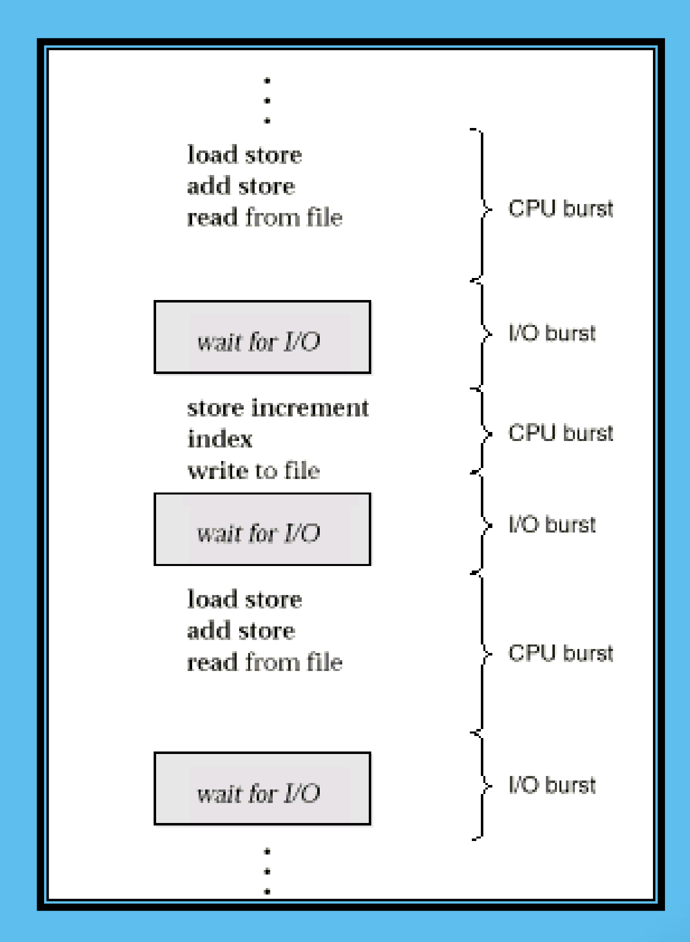
- La multiprogrammation est conçue pour obtenir une utilisation maximale des ressources, surtout l'UCT

- L'ordonnanceur UCT est la partie du SE qui décide quel processus dans la file ready/prêt obtient l 'UCT quand elle devient libre

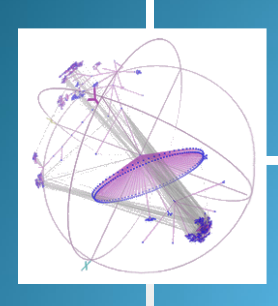
  - Un programme principalement E/S (interactif) a généralement de court pics d'activité
  - Un programme de calcul au contraire peut avoir des pics d'activité très longs.

Alternance CPU E/S





Cycles (bursts) d'UCT et E/S: l'exécution d'un processus consiste de séquences d'exécution sur UCT et d'attentes E/S



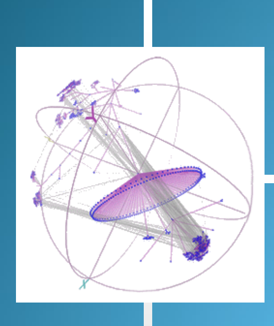
Quand invoquer l’ordonnanceur

- Choisir un processus parmi ceux qui sont prêts et lui donner les ressources CPU.

- L’ordonnancement a lieu quand un processus :

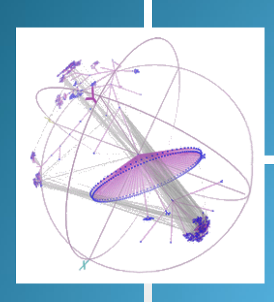
  1. Se termine.
  2. Passe de l’état “actif” à “attente”.
  3. Passe de l’état “actif” à “prêt”.
  4. Passe de l’état “attente” à “prêt”.

Un nouveau processus doit être choisi pour 1 et 4
Pour 2 et 3 : mode préemptif ou non préemptif.



Critères d’ordonnancement

- Il y a normalement plusieurs processus dans la file des prêt
- Quand l’UCT devient disponible, lequel choisir?
- L’idée générale est d’effectuer le choix dans l’intérêt de l’efficacité d’utilisation de la machine
- Mais cette dernière peut être jugée selon différents critères…



Critères de qualité

- Utilisation UCT: pourcentage d'utilisation
- Débit = Throughput: nombre de processus terminés dans une unité de temps
- Temps de rotation, temps de virement temps d'exécution, turnaround: le temps pris par le processus de son arrivée à sa terminaison.
  - inclus : temps passé dans tous les états, à faire des E/S...
- Temps d'attente: temps passé dans l'état "prêt" (somme de tout les temps passés dans la file des prêt)
- Temps de réponse : temps passé entre la création et le premier passage actif
  - Important pour l'interactivité

Mesures très liées.



Critères d’optimisation

- Exemples :
  - Maximiser l’utilisation et le débit
  - Minimiser le temps d’exécution, d’attente et de réponse

- Approches classiques :
  - En général : optimiser les valeurs moyennes.
  - Parfois : optimiser les valeurs min-max :
    - minimiser le temps maximal de réponse



Ordonnancement PAPS (FIFO)

Premier-arrivé Premier-servi (First-in First-out)

- Les processus prêt sont stockés dans une FIFO et servis par ordre d'arrivée
- L'ordonnancement PAPS est non préemptif
  - Mauvais partage du temps

Processus	Temps d'exécution
P_1	24
P_2	3
P_3	3



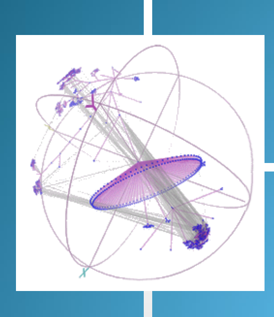
Un exemple pour PAPS

- Ordre d'arrivée : P_1 , P_2 , P_3
- Diagramme de Gantt :



- Temps d’attente : $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Temps moyen d’attente : (0+24+27)/3=17

Un autre exemple

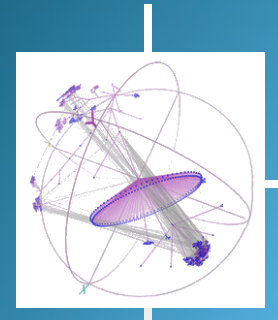


- Ordre d'arrivée : P_2 , P_3 , P_1
- Diagramme de Gantt :



- Temps d’attente : $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Temps moyen d’attente : (6+0+3)/3 = 3

Bien meilleur que dans le cas précédent.



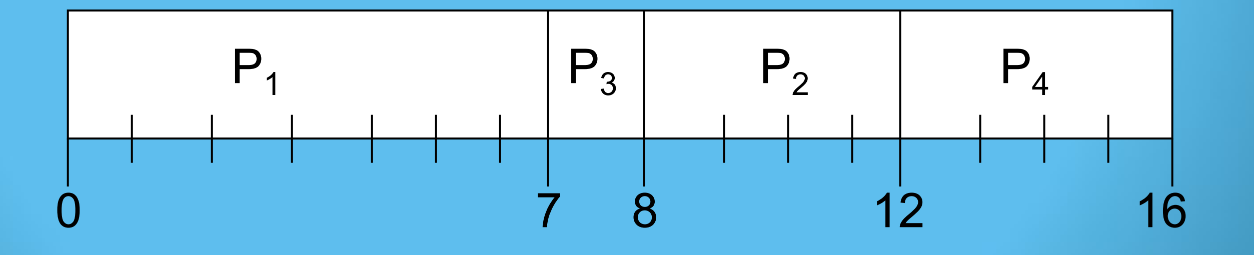
Plus court temps d’exécution (PCTE) Shortest job first (SJF)

- Le CPU est attribué au processus qui a le plus petit temps d’exécution (en utilisant PAPS en cas d’égalité)
- Deux approches :
  - Non préemptif (PCTE, SJF) : quand le CPU est accordé, il ne change pas jusqu’à la fin de son utilisation.
  - Préemptif (PCTER, SRTF) : si un nouveau processus arrive avec un temps d’exécution plus court que ce qui reste au processus courant il prend sa place
- PCTER : optimal pour le temps d’attente moyen.



Exemple pour PCTE

Processus	Arrivée	Durée
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

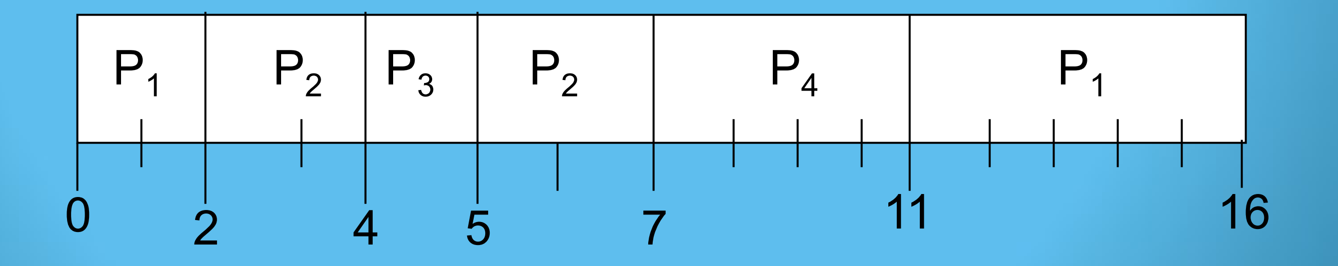


- Temps moyen d'attente = (0+6+3+7)/4=4



Exemple pour PCTER

Processus	Arrivée	Durée
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



- Temps moyen d'attente = (9+1+0+2)/4=3



Prédire la durée d'utilisation

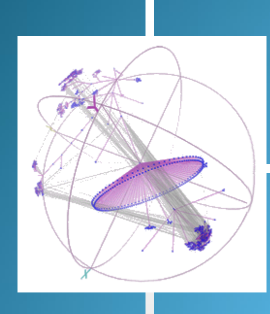
- Utilisation d'une moyenne des durées précédentes

1. $t_n =$ durée du $n^{ième}$ pic de CPU
2. $\tau_{n+1} =$ durée prédite du prochain pic
3. $\alpha, 0 \leq \alpha \leq 1$

Alors :

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n .$$

t_n : histoire la plus récente, τ_n : histoire passée



Moyennage exponentiel

$\alpha=0$: $\tau_{n+1}=\tau_{n}$: le passé récent ne compte pas.

$\alpha=1$: $\tau_{n+1}=t_{n}$: seul le passé récent compte.

- En général, $\alpha = 1/2$
- τ_0 est une constante (moyenne du système)
- La formule peut être développée en :

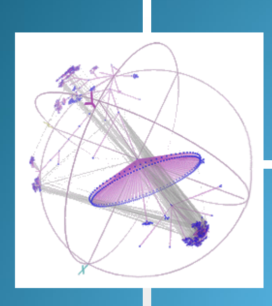
  $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots + (1 - \alpha)^j\, \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1}\, \tau_0$

- Comme α et $(1 - \alpha)$ sont plus petits que 1, les termes ont de moins en moins de poids.



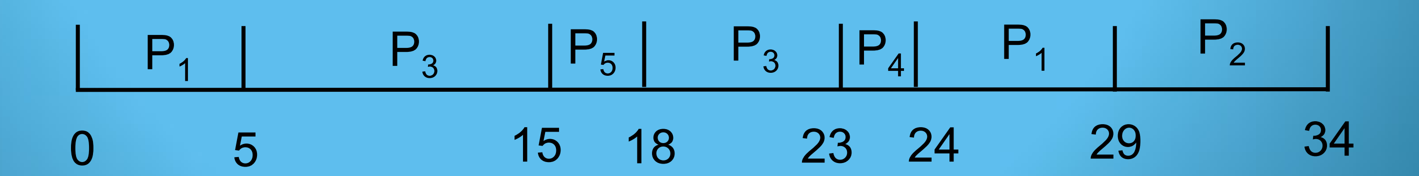
Ordonnancer avec priorités

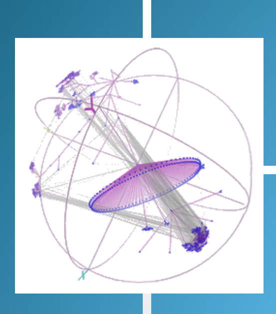
- Une priorité (entière) est associée à chaque processus
- Le CPU est donné au processus le plus prioritaire (avec PAPS si besoin)
  - Préemptif : si un nouveau processus plus prioritaire arrive, lui donner la main
  - Non préemptif : le processus courant termine son cycle
- PCTE est un algorithme à priorité.
- Risque de famine : augmenter la priorité avec l'âge: technique de vieillissement.



Priorité + préemption

Processus	Durée	Arrivée	Priorité
P_1	10	0	3
P_2	5	2	7
P_3	15	5	2
P_4	1	10	2
P_5	*3*	*15*	*1*





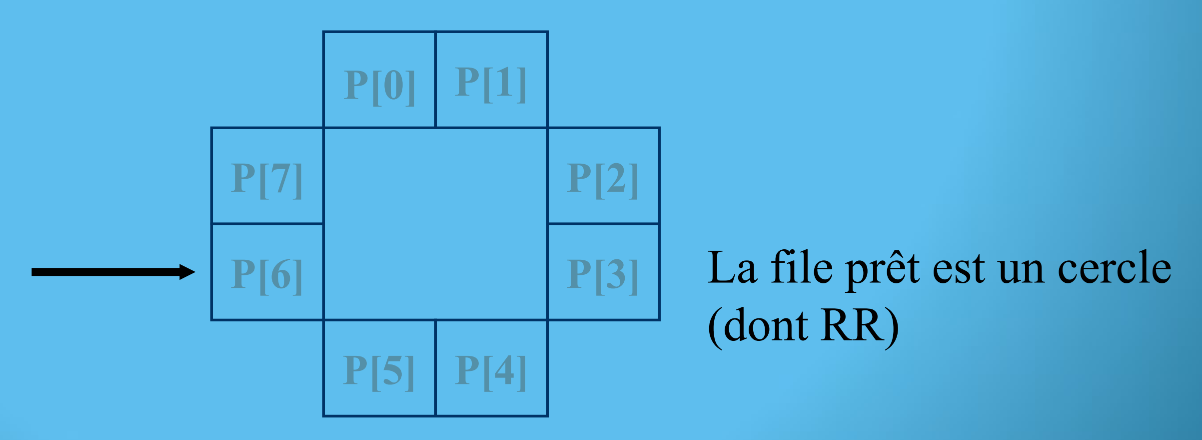
Tourniquet (Round Robin)

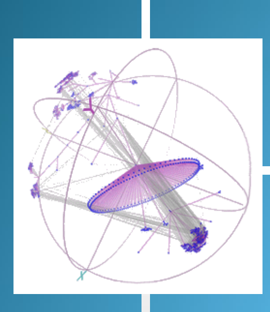
- Initialement pour les systèmes à temps partagé.
- Chaque processus obtient un peu de temps CPU (un *quantum*), typiquement de 10-100 millisecondes puis est préempté et devient “prêt”.
  - Similaire à PAPS avec préemption
- S’il y a n processus prêt avec un quantum q, alors le temps de réponse est au maximum $(n-1)q$



Tourniquet (Round Robin)

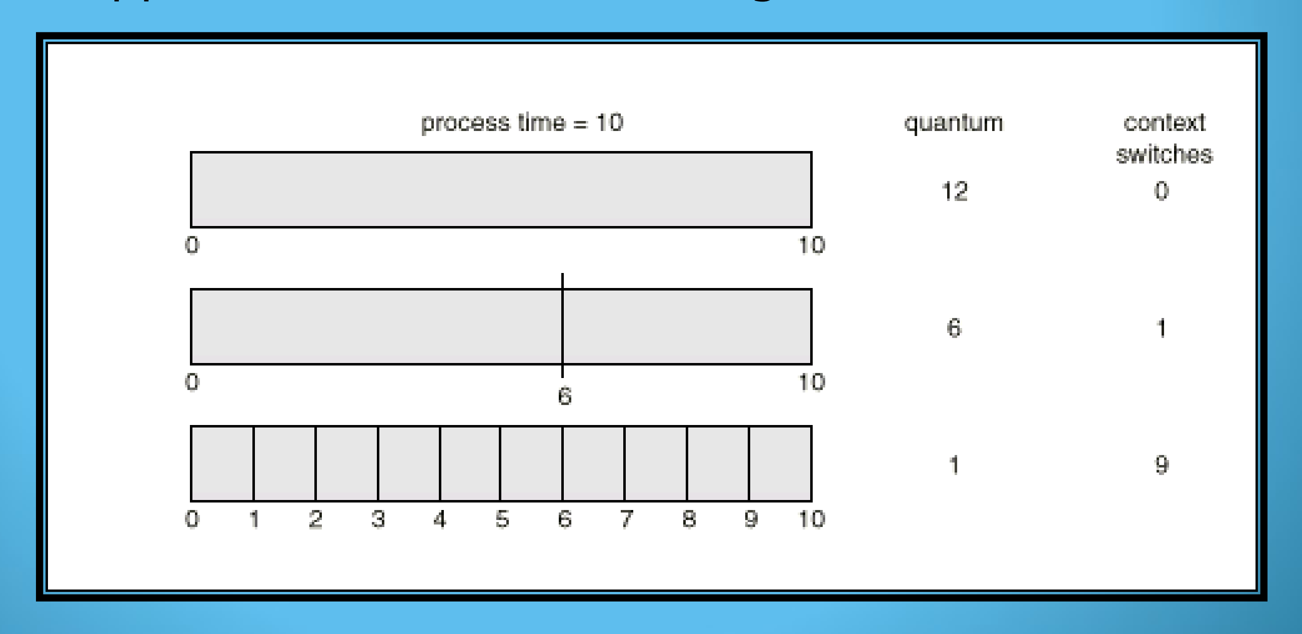
- S’il s’exécute pour un quantum entier sans autres interruptions, il est interrompu par la minuterie et l’UCT est donnée à un autre processus
- Le processus interrompu redevient prêt (à la fin de la file)

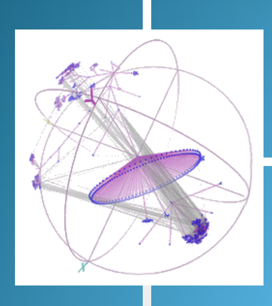




Changements de contexte

- La performance dépend de q
  - *q* grand : similaire à PAPS
  - Si q est petit, le temps de réponse diminue mais il y a plus de changements de contexte. q doit être grand par rapport à la durée d'un changement de contexte

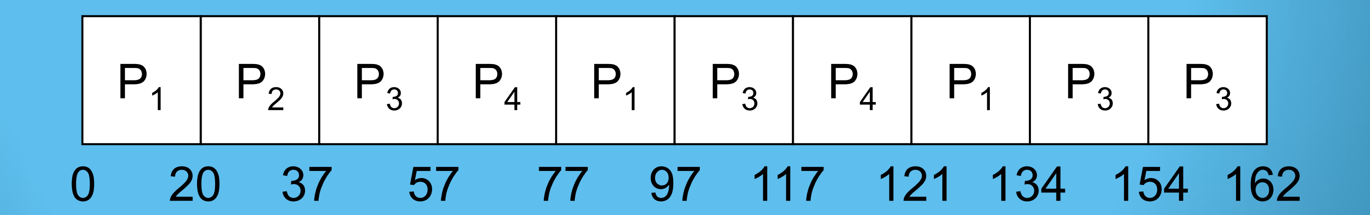




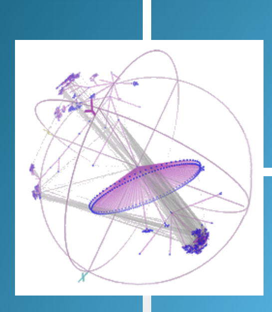
Tourniquet avec q = 20

Processus	Durée
P_1	53
P_2	17
P_3	68
P_4	24

- Le diagramme de Gantt est :

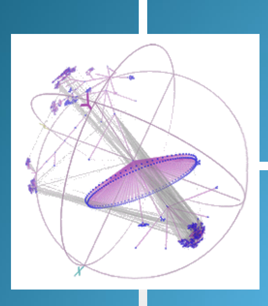


- Typiquement temps d'exécution supérieur à PCTE mais meilleur temps de réponse.



Ordonnancement multi-files

- La file pour les “prêts” est séparée en plusieurs files
  - distinguer différents types de processus.
- Un algorithme pour chaque file
- L’ordonnancement obligatoire entre les files
  - Une file prioritaire avec préemption : risque de famine.
  - Partage du temps : chaque file obtient une fraction du temps qu’elle gère comme elle le souhaite



Exemple d'ordonnancement

highest priority

system processes

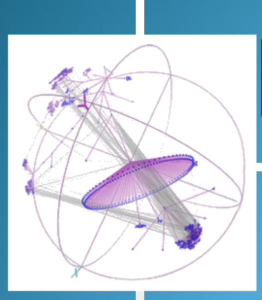
interactive processes

interactive editing processes

batch processes

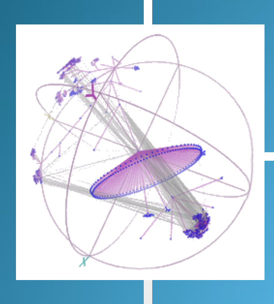
student processes

lowest priority



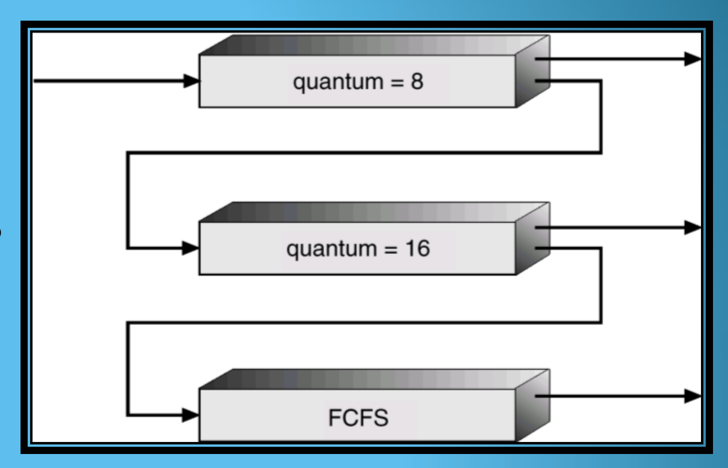
Plusieurs files avec feedback

- Un processus peut changer de file
  - Gestion de l'âge par exemple.
- Ce type d’ordonnanceur est défini par :
  - nombre de files,
  - ordonnanceur pour chaque file
  - règles pour changer un processus de file (vers le haut ou vers le bas)
  - règles pour décider de la file initiale d’un processus
- Algorithme d’ordonnancement le plus complexe et le plus général

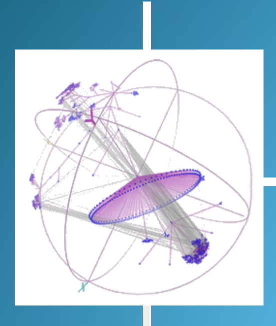


Exemple

- Trois files :
  - Q_0 – tourniquet, q=8 ms
  - Q_1 – tourniquet, q=16 ms
  - Q_2 – PAPS



- Ordonnancement
  - Arrivée dans Q_0 avec une règle PAPS, puis tourniquet de 8 secondes. S'il n'a pas terminé en 8 millisecondes il passe dans Q_1.
  - Dans Q_1, la règle est toujours PAPS et le processus obtient 16 millisecondes. S'il n'a toujours pas terminé, il est préempté et passe dans Q_2.
  - Règles inter files...



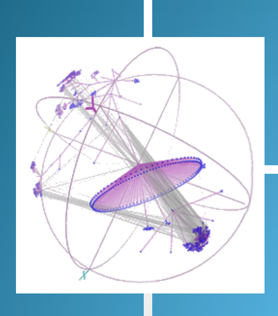
Évaluation des algorithmes

- Comment choisir un algorithme ?

1. Choix des critères (utilisation du CPU maximale avec temps de réponse < 1s)
2. Évaluer/comparer les algorithmes

- Méthodes d’évaluation

1. Modélisation déterministe (scénario)
2. Modélisation des files d’attente
3. Simulation (scénario aléatoire ou rejoué)
4. Implémentation dans le SE (très coûteux)



En pratique…

- Les méthodes que nous avons vu sont toutes utilisées en pratique (sauf plus court servi *pur* qui est impossible)
- Les SE sophistiqués fournissent au gérant du système une librairie de méthodes, qu'il peut choisir et combiner au besoin après avoir observé le comportement du système
- Pour chaque méthode, plusieurs paramètres sont disponibles: p.ex. durée du quantum, etc.



Ordonnancement Linux

- Ordonnancement temps partagé
  - Seuls les processus en mode utilisateur et non en mode superviseur peuvent être préemptés
  - priorité avec des crédits (pour celui qui en a le plus)
  - Quand le quantum expire, perte de 1 crédit (vieillissement)
  - Un processus sans crédit est suspendu
  - Si tous les processus prêt sont suspendus, tous les processus en regagnent
  - Les processus en arrière plan on une priorité plus faible et reçoivent donc moins de crédits.

  $\Rightarrow$ Processus interactifs accumulent plus de crédits



Systèmes d'exploitation
Chapitre III

Gestion de la Mémoire



Objectifs

Organisation de la mémoire principale :

- Savoir quelles zones sont libres et quelles zones sont utilisées.
- Règles d'allocation : qui obtient de la mémoire, combien, quand, etc.
- Techniques d'allocation – choix des lieux d'allocation et mises à jour des informations d'allocation.
- Règles de désallocation : à qui reprendre de la mémoire, quand, etc.



Principaux critères

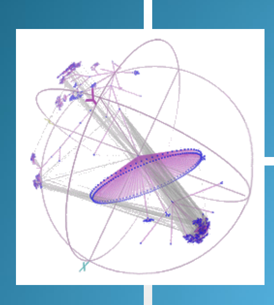
- Protection : plusieurs programmes/utilisateurs en mémoire simultanément. Il faut donc éviter les interférences.
- Partitionnement statique ou dynamique de la mémoire.
- Placement des programmes dans la mémoire : où, qui enlever si la mémoire est pleine ? Partage de code ou de données…
- Le partage doit être transparent : l'utilisateur ne doit pas savoir si la mémoire est partagée ou pas (illusion d'une mémoire infinie).



Cas mono-processus

- Système de gestion simple
- La mémoire est divisée entre le SE et les processus utilisateur.
- Le SE est protégé des programmes de l'utilisateur (pour éviter que l'utilisateur n'écrive sur le code du SE)

SE
Processus utilisateur
Espace libre



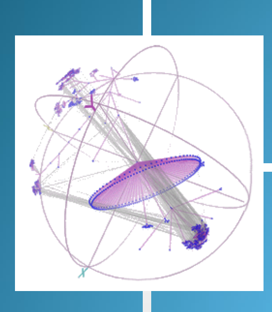
Cas mono-processus

Avantages :

- Simplicité
- SE très réduit

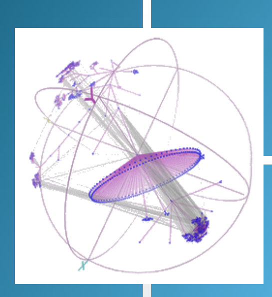
Inconvénients :

- Mauvaise utilisation de la mémoire (un seul processus) et du processeur (attente des E/S).
- Manque de flexibilité : les programmes sont limités à la mémoire existante.



Cas multi-utilisateur

- Plus d'un processus à la fois.
  - Maximiser le degré de multiprogrammation.
  - Meilleure utilisation de la mémoire et du CPU.
- Deux problèmes :
  - Réallocation.
  - Protection.

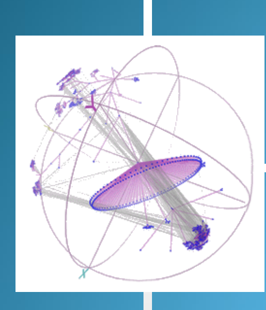


Adresses physiques et logiques

- Mémoire physique:
  - la mémoire principale RAM de la machine
- Adresses physiques:
  - les adresses de cette mémoire
- Mémoire logique:
  - l'espace d'adressage d'un programme
- Adresses logiques:
  - les adresses dans cet espace

- Il faut séparer ces concepts car normalement, les programmes sont à chaque fois chargés à des positions différentes dans la mémoire

Donc adresse physique ≠ adresse logique



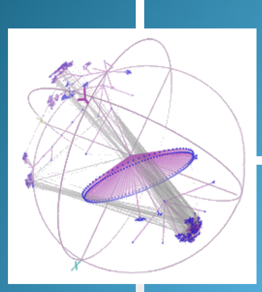
Adresses physiques et logiques

- Une adresse logique est une adresse à une location de programme
    - par rapport au programme lui-même seulement
    - indépendante de la position du programme en mémoire physique
- Plusieurs types d'adressages p.ex.
    - les adresses du programmeur (noms symboliques) sont traduites au moment de la compilation dans des
        - adresses logiques
        - ces adresses sont traduites en adresses physiques après chargement du programme en mémoire par l'unité de traduction adresses (MMU)



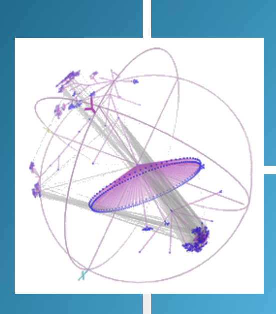
Liaison(Binding) d'adresses logiques et physiques

- La liaison des adresses logiques aux adresses physiques peut être effectuée à des moments différents:
  - Compilation: quand l'adresse physique est connue au moment de la compilation (rare)
    - p.ex. parties du SE
  - Chargement: quand l'adresse physique où le programme est chargé est connue, les adresses logiques peuvent être traduites (rare aujourd'hui)
  - Exécution: normalement, les adresses physiques ne sont connues qu'au moment de l'exécution
    - p.ex. allocation dynamique



Traduction d'adresses logiques en adresses physiques

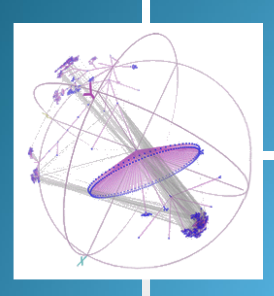
- Dans les premiers systèmes, un programme était toujours lu aux mêmes adresses de mémoire
- La multiprogrammation et l'allocation dynamique ont donc engendré le besoin de lire un programme dans des positions différentes
- Au début, ceci était fait par le chargeur (loader) qui changeait les adresses avant de lancer l 'exécution
- Aujourd'hui, ceci est fait par le MMU au fur et à mesure que le programme est exécuté
- Ceci ne cause pas d'hausse de temps d'exécution, car le MMU agit en parallèle avec autres fonctions d'UCT
  - P.ex. l'MMU peut préparer l'adresse d'une instruction en même temps que l'UCT exécute l'instruction précédente



Protection et Réallocation

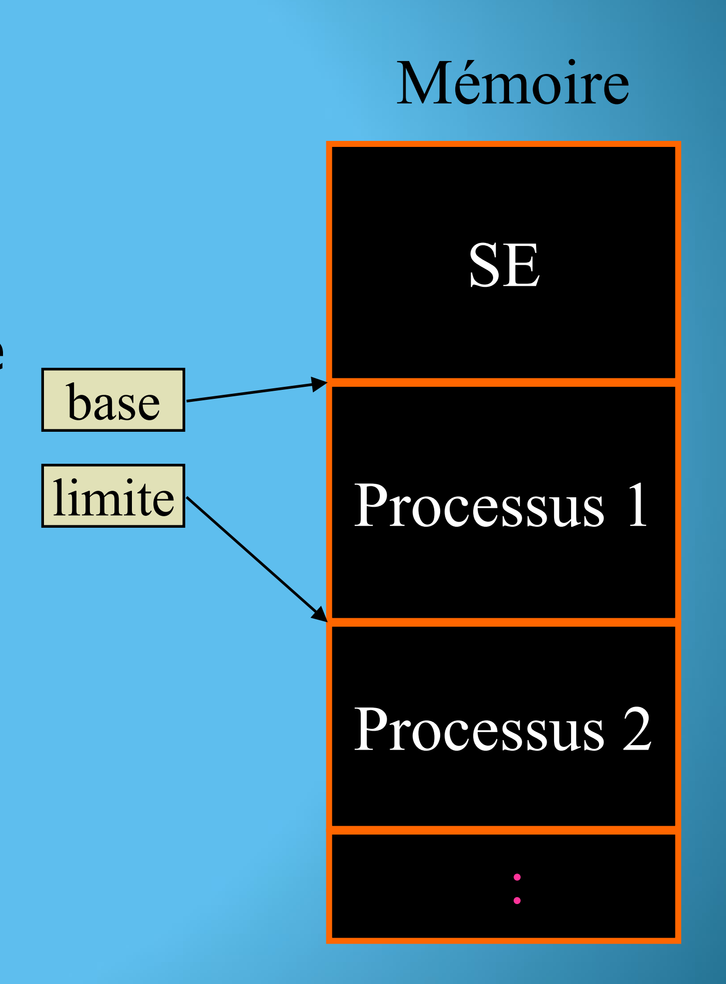
- **Protection :** empêcher qu'un processus ne puisse écrire dans l'espace d'adressage d'un autre processus.

- **Réallocation :** si le même code est placé à différents endroits de la mémoire cela peut poser des problèmes si l'adressage est absolu :



Solution

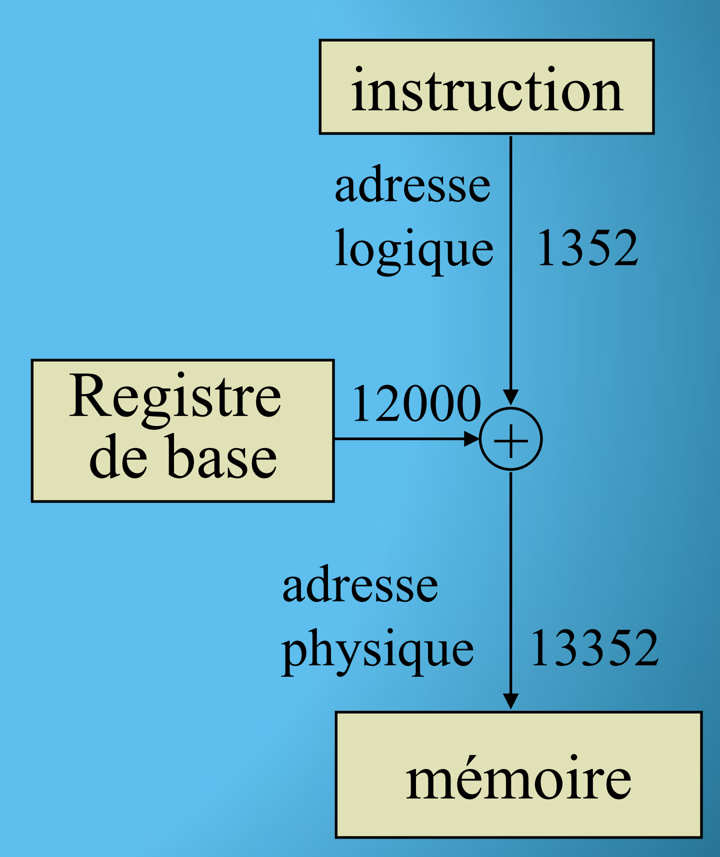
- Avec les registres base et limite, les deux problèmes sont résolus :
- Le registre base stocke l'adresse du début de la partition et le registre limite sa taille.
- Toute adresse générée doit être inférieure à limite et est ajoutée au registre base.
- Si un processus est déplacé en mémoire, il suffit alors de modifier son registre base.

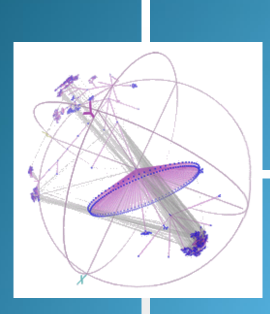




Adresses logiques/physiques

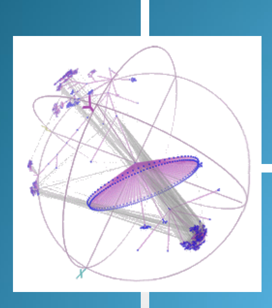
- Le concept d'adressage logique distinct de la zone d'adressage physique est central pour les systèmes de gestion de la mémoire.
  - *Adresse logique ou virtuelle* – adresse générée par le CPU.
  - *Adresse physique* – adresse utilisée par le périphérique mémoire.





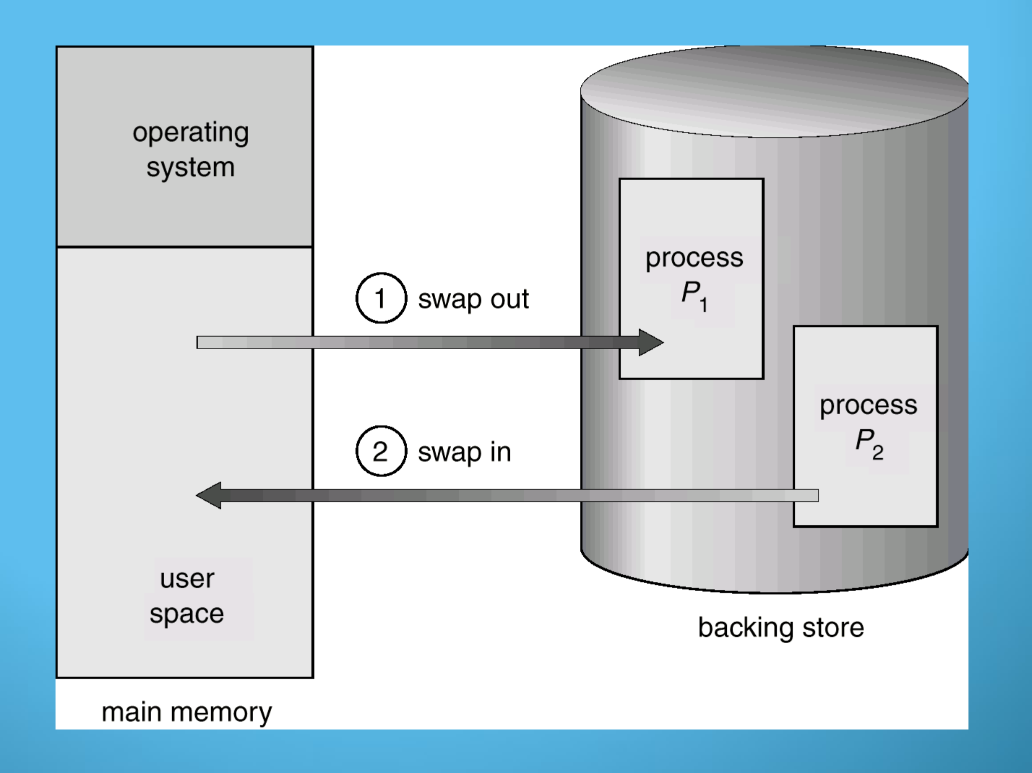
Unité de gestion de la mémoire

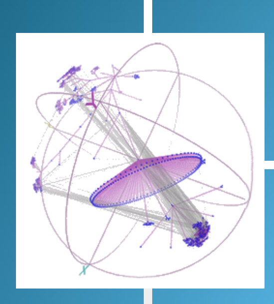
- L'unité de gestion de la mémoire (MMU) est un périphérique qui converti les adresses logiques en adresses physiques.
- Avec un MMU, le registre base est appelé registre de réallocation et il est ajouté à toute adresse générée par un processus au moment de l'envoi vers la mémoire physique.
- Quand l'ordonnanceur choisi un programme, les registres de réallocation et limite sont mis à jour.
- L'utilisateur n'utilise jamais d'adresse physique.



Swapping

- Un processus peut être sorti de la mémoire et stocké dans une zone annexe puis rapatrié plus tard pour terminer son exécution.





Swapping (2)

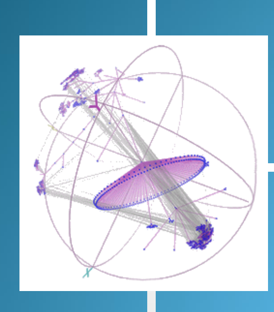
Changement de contexte : exemple
Processus = *100 Ko*
Stockage à un taux de 1Mo/s
Transfert = 100 ms
+ latence de 8 ms, temps total de swap : 216 ms.

- Pour bien utiliser le CPU, il faut que le temps d'exécution soit long par rapport au temps de swap. Tourniquet : q > 216 ms.
- Le swap standard est trop coûteux et peu utilisé en pratique.
- Des versions modifiées sont utilisées sur la plupart des systèmes (UNIX, Windows, etc.).



Techniques de gestion Méthodes d’allocation

- Pour un espace donné on peut choisir deux modes d’allocation :
- **Allocation contiguë :** consiste à placer la totalité d’un programme à des adresses consécutives
- **Allocation non contiguë :** consiste à fractionner le programme et à placer les différents fragments à des adresses dispersées



Techniques de gestion

Divers types de gestion de la mémoire ont été utilisés.

- Dans l'allocation contiguë
  - Partitionnement statique
    - Partitions fixes
  - Partitionnement dynamique
    - Partitions variable
- Dans l'allocation non contiguë
  - Segmentation
  - Pagination simple
  - Pagination à la demande

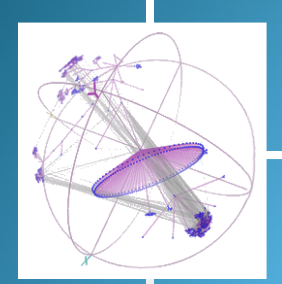
Les systèmes récents utilisent la mémoire virtuelle



Partitionnement statique

- Méthode la plus simple :
  - La mémoire est partagé en plusieurs partitions de la même taille.
  - Chaque partition peut contenir un processus (limite du nombre de processus au nombre de partitions).
  - Quand une partition est libérée, un autre processus est choisi.

Partitionnement statique
Partitions fixes



Première organisation de l'allocation contiguë

- Mémoire principale subdivisée en régions distinctes: partitions
- Les partitions sont soit de même taille ou de tailles inégales
- N'importe quel programme peut être affecté à une partition qui soit suffisamment grande

Operating System
8 M
8 M
8 M
8 M
8 M
8 M
8 M
8 M

Equal-size partitions

Operating System
8 M
2 M
4 M
6 M
8 M
8 M
12 M
16 M

Unequal-size partitions

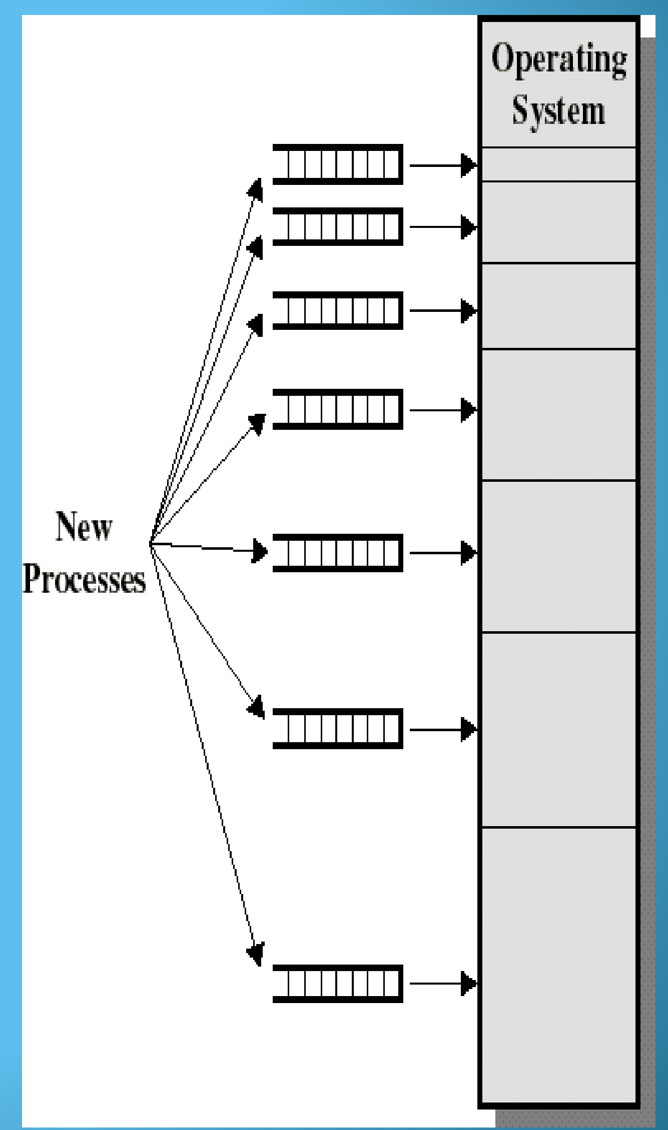
Algorithme de placement partitions fixes

Partitions de tailles inégales: utilisation de plusieurs files d'attentes

- assigner chaque processus à la partition de la plus petite taille pouvant le contenir
- Une file par taille de partition
  - tente de minimiser la fragmentation interne

Problème:

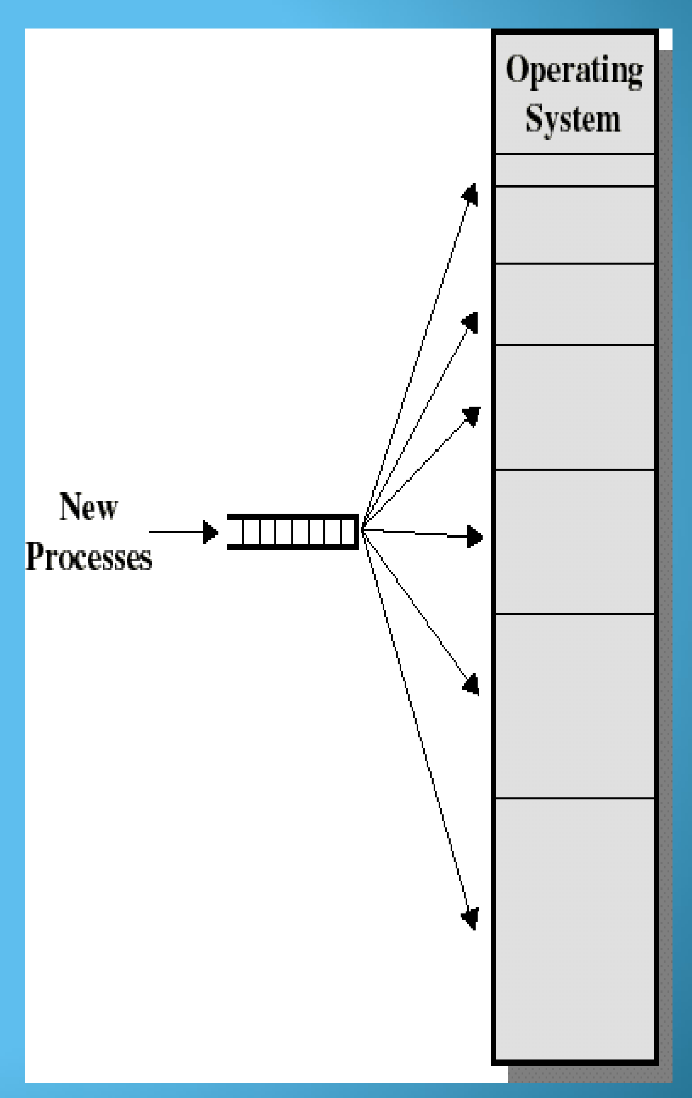
certaines files seront vides s'il n'y a pas de processus de cette taille (fragemantation externe)

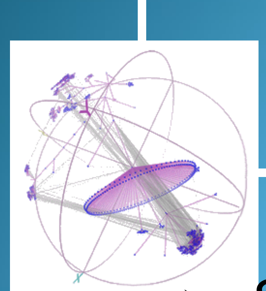


Algorithme de placement partitions fixes

Partitions de tailles inégales: utilisation d'une seule file

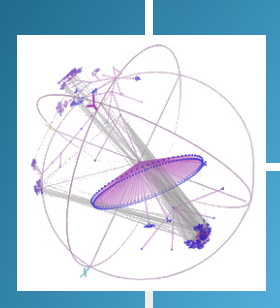
- On choisit la plus petite partition libre pouvant contenir le prochain processus
- le niveau de multiprogrammation augmente au profit de la fragmentation interne





Partitions fixes

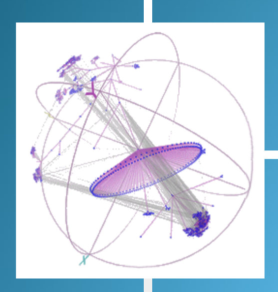
- Simple, mais...
  - Inefficacité de l'utilisation de la mémoire: tout programme, si petit soit-il, doit occuper une partition entière. Il y a fragmentation interne.
  - Les partitions à tailles inégales atténue ces problèmes mais ils y demeurent...

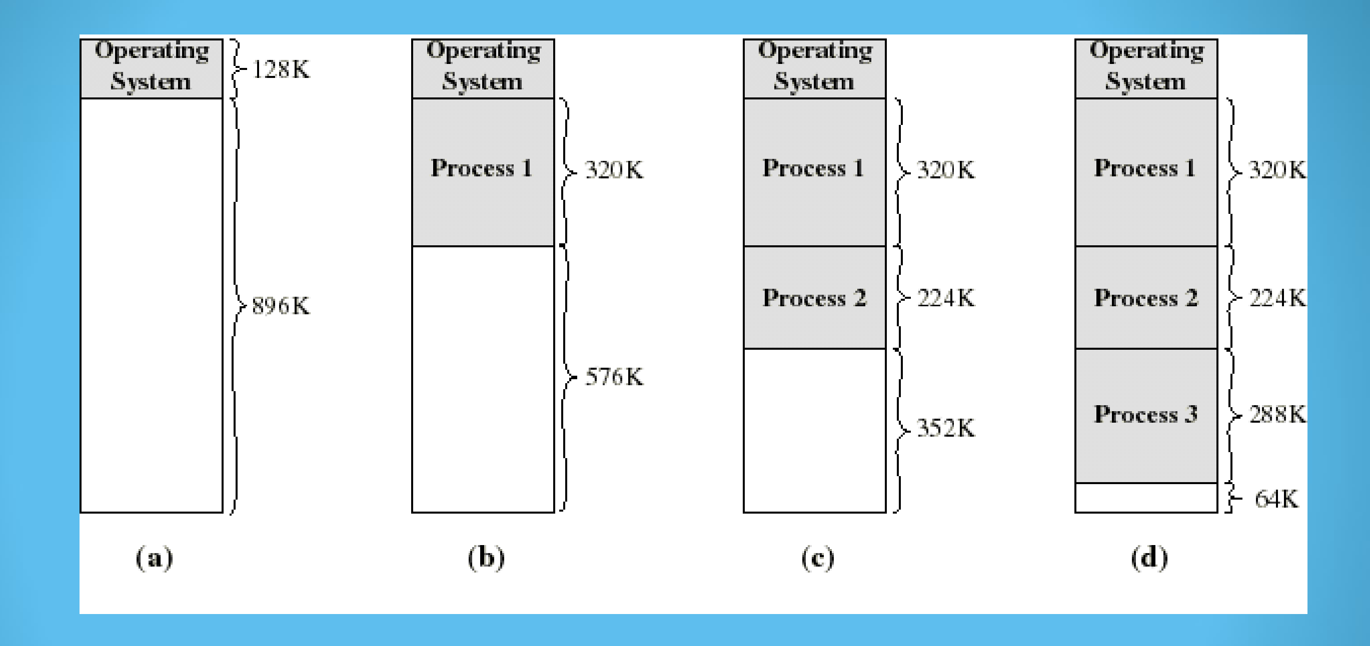


Partitions dynamiques

- Partitions en nombre et tailles variables
- Chaque processus est alloué exactement la taille de mémoire requise
- Probablement des trous inutilisables se formeront dans la mémoire: c’est la fragmentation externe

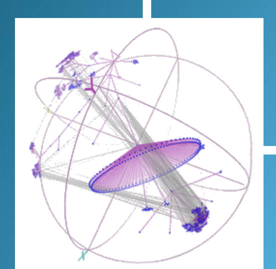
Partitions dynamiques

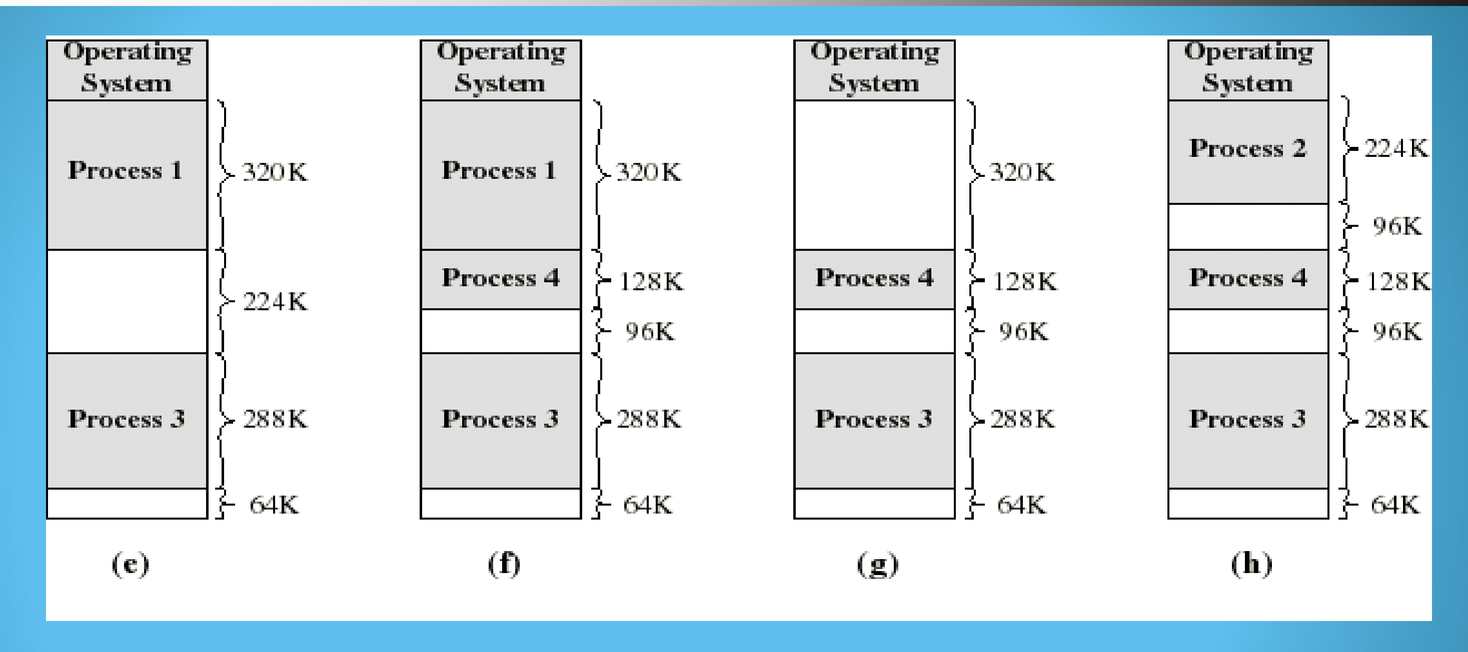




- (d) Il y a un trou de 64K après avoir chargé 3 processus: pas assez d'espace pour un autre processus
- Si tous les processus se bloquent (p.ex. attente d'un événement), P2 peut être permuté et P4=128K peut être chargé.

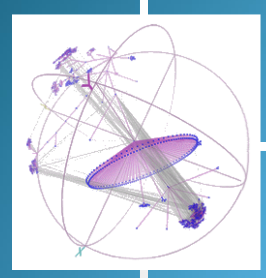
Partitions dynamiques





- (e-f) P2 est suspendu, P4 est chargé. Un trou de 224-128 =96K est créé (fragmentation externe)
- (g-h) P1 se termine ou il est suspendu, P2 est chargé à sa place: produisant un autre trou de 320-224=96K...
- Nous avons 3 trous petits et probablement inutiles. 96+96+64=256K de fragmentation externe
- COMPRESSION pour en faire un seul trou de 256K

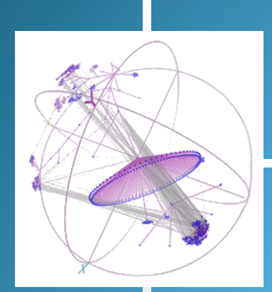
Gestion de la mémoire



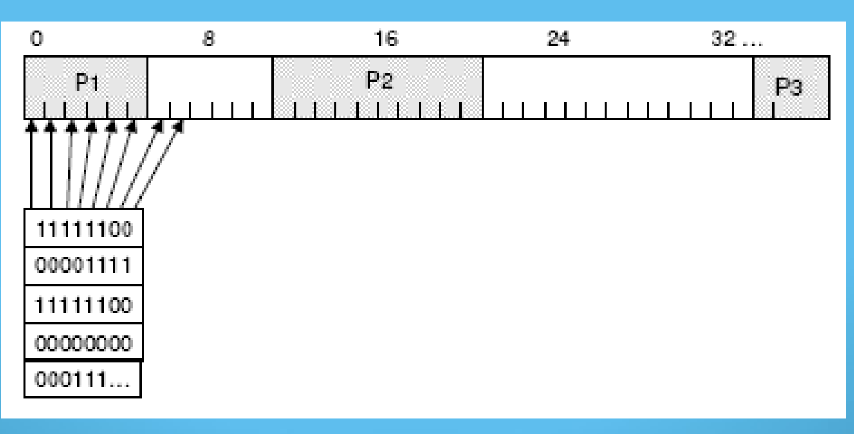
- Le système garde la trace des emplacements occupés de la mémoire par l'intermédiaire :
  - D'une table de bits ou bien
  - D'une liste chaînée.

La mémoire étant découpée en unités, en blocs, d'allocation

État de la mémoire *Tables de bits*

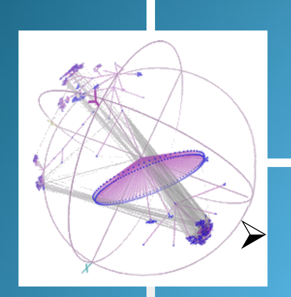


- On peut conserver l'état des blocs de mémoire grâce à une table de bits. Les unités libres étant notées par 0 et ceux occupées par un 1. (ou l'inverse).
- La technique des tables de bits est simple à implanter, mais elle est peu utilisée.
- plus l'unité d'allocation est petite, moins on a de pertes lors des allocations, mais en revanche, plus cette table occupe de la place en mémoire.

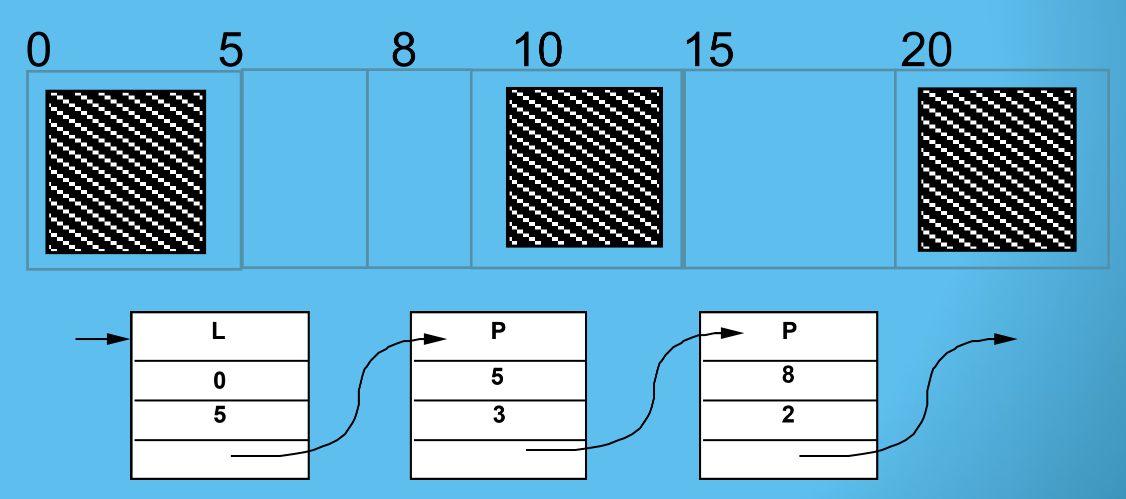


État de mémoire avec trois processus et son bitmap

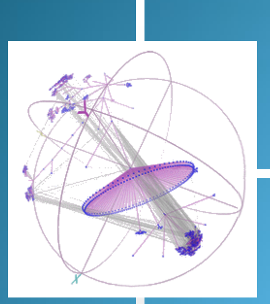
Gestion de la mémoire
Listes chaînées



- On peut représenter la mémoire par une liste chaînée de structures dont les membres sont :
  - le type (libre ou occupé),
  - l'adresse de début,
  - la longueur, et
  - un pointeur sur l'élément suivant.



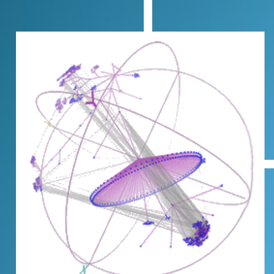
- On peut modifier ce schéma en utilisant deux listes
  - une pour les processus et
  - l'autre pour les zones libres.



Allocation d'espace mémoires

- L'allocation d'un espace libre pour un processus peut se faire suivant quatre stratégies principales:
  - First-fit (1ère zone libre) : allouer le premier trou de taille suffisante dans la liste.
  - Next-fit (Zone libre suivante) : La recherche suivante commencera à partir de la position courante et non à partir du début.
  - Best-fit (Meilleur ajustement) : allouer le plus petit trou qui soit suffisamment grand (garder la liste triée par taille).
  - Worst-fit (pire ajustement): allouer le plus grand trou.

Algorithmes de Placement Exemple



8K
12K
22K
18K
Last allocated block (14K)
8K
6K
14K
36K

(a) Before

First Fit
Best Fit
Worst Fit

8K
12K
6K
2K
8K
6K
14K
20 K

(b) After

Allocated block
Free block

Example Memory Configuration Before and After Allocation of 16 Kbyte Block

Algorithmes de placement

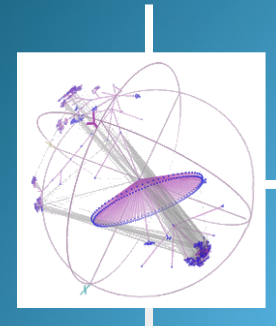
- Quel est le meilleur?

critère principal: diminuer la probabilité de situations où un processus ne peut pas être servi, même s'il y a assez de mémoire...

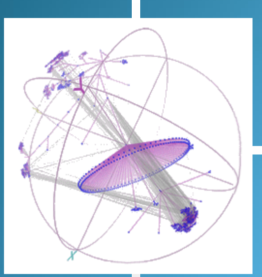
- ✓ La simulation montre qu'il ne vaut pas la peine d'utiliser les algorithmes les plus complexes
- ✓ "Best-fit": cherche le plus petit bloc possible: l'espace restant est le plus petit possible
- ✓ la mémoire se remplit de trous trop petits pour contenir un programme
- ✓ "Worst-fit": les allocations se feront souvent à la fin de la mémoire et donc trop de temps

... D'où le first fit

Fragmentation mémoire non utilisée

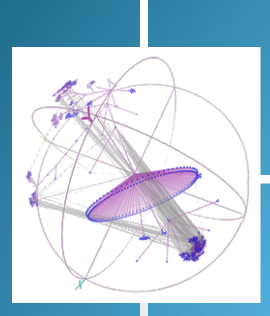


- Un problème majeur dans l'affectation contiguë:
  - Il y a assez d'espace pour exécuter un programme, mais il est fragmenté de façon non contiguë
    - externe: l'espace inutilisé est entre partitions
    - interne: l'espace inutilisé est dans les partitions



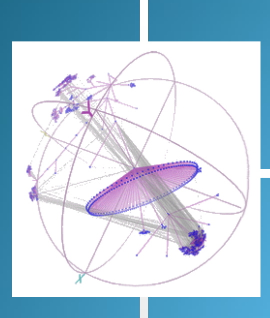
Compaction

- Une solution pour la fragmentation externe
- Les programmes sont déplacés en mémoire de façon à réduire à 1 seul grand trou plusieurs petits trous disponibles
- Effectuée quand un programme qui demande d'être exécuté ne trouve pas une partition assez grande, mais sa taille est plus petite que la fragmentation externe existante
- Désavantages:
  - temps de transfert programmes
  - besoin de rétablir tous les liens entre adresses de différents programmes



Allocation non contiguë

- A fin réduire le besoin de compression, le prochain pas est d'utiliser l'allocation non contiguë
  - diviser un programme en morceaux et permettre l'allocation séparée de chaque morceau
  - les morceaux sont beaucoup plus petits que le programme entier et donc permettent une utilisation plus efficace de la mémoire
    - les petits trous peuvent être utilisés plus facilement

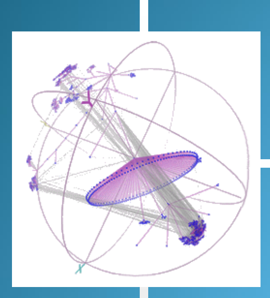


Allocation non contiguë

- Il y a deux techniques de base pour faire ceci: la pagination et la segmentation

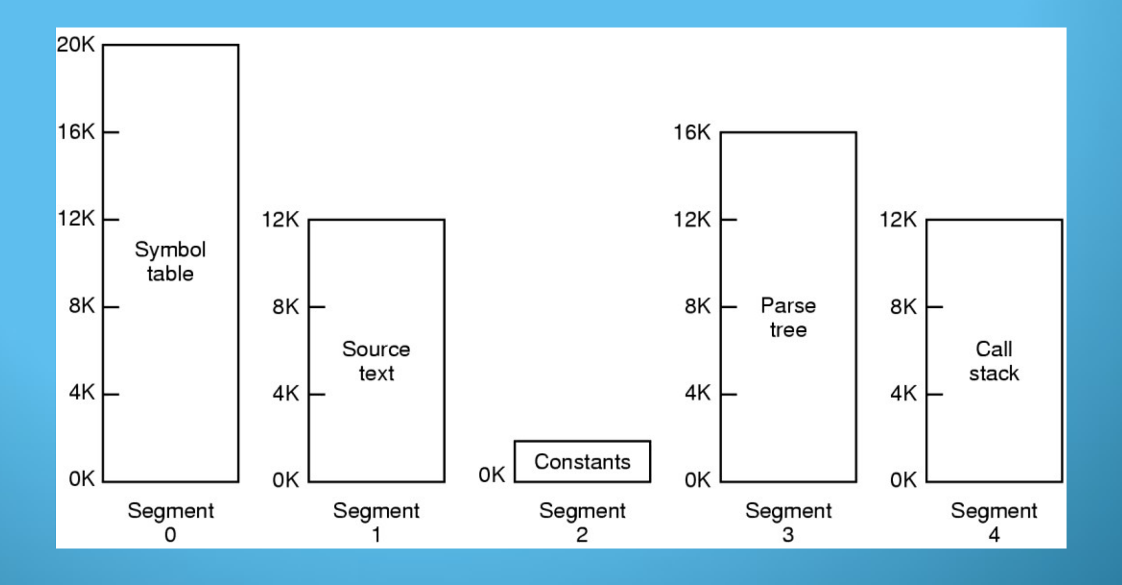
  - la segmentation utilise des parties de programme qui ont une valeur logique (des modules)
  - la pagination utilise des parties de programme arbitraires (morcellement du programmes en pages de longueur fixe).

  - La combinaison des deux



Segmentation

- Segmentation : plusieurs espaces d'adressage distincts.
  - L'espace logique correspond alors à un ensemble de segments de taille variable indépendants.
  - Les segments peuvent croître ou diminuer dynamiquement et indépendamment des autres segments.





Segmentation

- Pour un processus, chaque module ou structure de données occupe un segment différent.
- Chaque segment a un nom/numéro et une longueur.
- Chaque adresse est définie par un numéro de segment et un décalage.
- Le compilateur crée les segments de manière automatique.

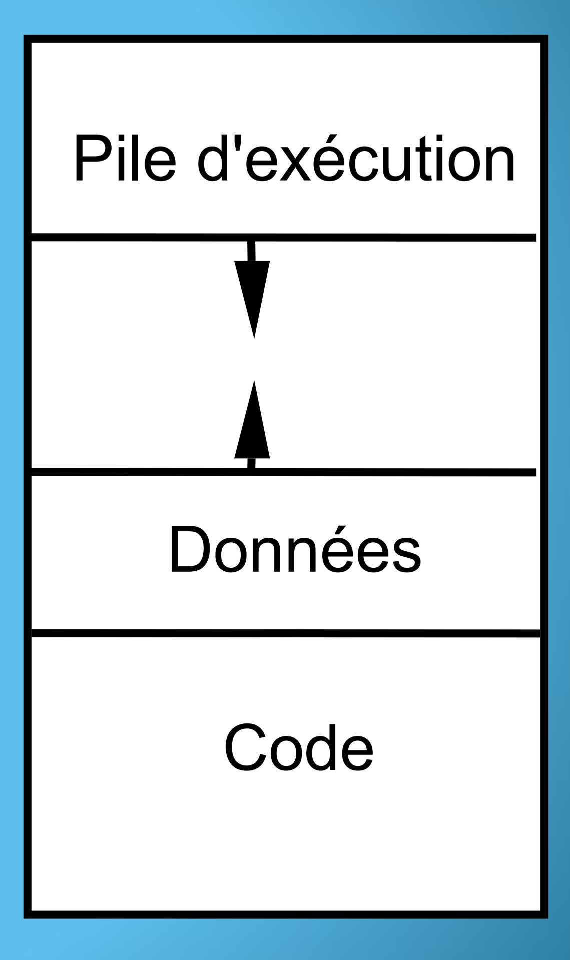
Avantages :

- La taille des segments est variable.
- Permet d'utiliser des structures dynamiques.
- Protection : les segments peuvent être associés à différents niveaux de privilèges.
- Code partagé plus simple à implémenter (segment partagé).

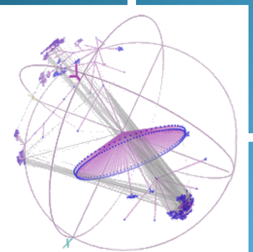


Espace de travail

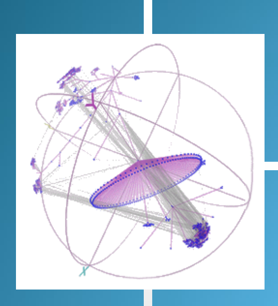
- Les processus sont composés d’un espace de travail en mémoire formé d’au moins trois segments



Les segments comme unités d'allocation mémoire



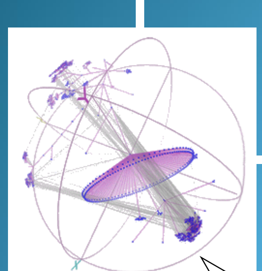
- Étant donné que les segments sont plus petits que les programmes entiers, cette technique implique moins de fragmentation (qui est externe dans ce cas)



Mécanisme de la segmentation

- Un tableau contient l'adresse de début de tous les segments dans un processus
- Chaque adresse dans un segment est ajoutée à l'adresse de début du segment par la MMU

Mécanisme de la segmentation

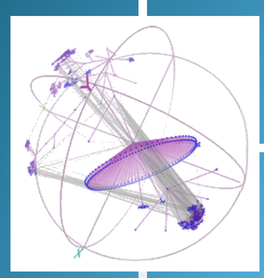


- L’adresse logique consiste el la paire:

  <No de segment, décalage>

  où décalage est l’adresse *dans* le segment
- le table des segments contient: les descripteurs de segments tels que:
  - adresse de base
  - longueur du segment
  - Informations de protection
- Dans le PBC du processus il y a:
  - un pointeur à l’adresse en mémoire de la table des segments
  - le nombre de segments dans le processus

Au moment de la commutation de contexte, ces infos seront chargées dans les registres appropriés d’UCT



Exemple de segmentation

emacs

segment 0

data 1

segment 1

logical memory
process P_1

	limit	base
0	25286	43062
1	4425	68348

segment table
process P_1

VI

segment 0

data 2

segment 1

logical memory
process P_2

	limit	base
0	25286	72773
1	8850	90003

segment table
process P_2

43062

editor

68348

data 1

72773

90003

data 2

98553

physical memory

Traduction d'adresses dans la segmentation



d

VI

Segment 0

Virtual Address

Seg #	Offset = d

Segment Table

Base + d

Register

Seg Table Ptr

Segment Table

S#

Length | Base

d

Segment

	limit	base
0	25286	72773
1	8850	90003

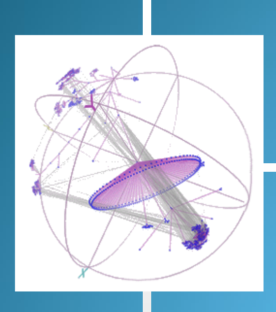
segment table process P_2

Program

Segmentation

Main Memory

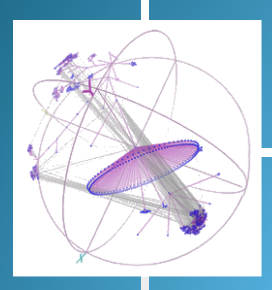
Aussi, si d > longueur: erreur!



Partage de segments

- Un des avantages : partage de code commun. Particulièrement important pour les systèmes à temps partagé où plusieurs utilisateurs partagent un programme (éditeur de texte, base de données, etc.)

- Code partagé
  - Une copie en lecture seule (code réentrant) est partagé entre plusieurs utilisateurs. Seul la position dans le code change selon les utilisateurs.
  - Le code partagé doit apparaître au même endroit dans l'espace logique de tous les processus.



Partage de segments

editor

segment 0

data 1

segment 1

logical memory
process P_1

	limit	base
0	25286	43062
1	4425	68348

segment table
process P_1

editor

segment 0

data 2

segment 1

logical memory
process P_2

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2

43062

editor

68348

data 1

72773

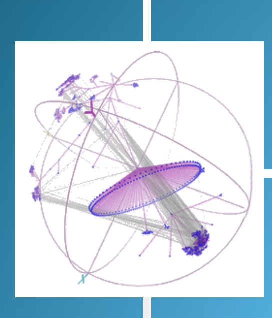
90003

data 2

98553

physical memory

P.ex: DLL utilisé par plus usagers

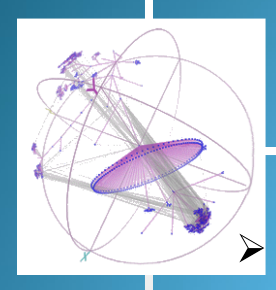


Segmentation et protection

- Chaque entrée dans la table des segments peut contenir des infos de protection:
  - longueur du segment
  - privilèges de l'usager sur le segment: lecture, écriture, exécution
    - Si au moment du calcul de l'adresse on trouve que l'usager n'a pas droit d'accès→interruption
    - ces infos peuvent donc varier d'un usager à autre, par rapport au même segment!

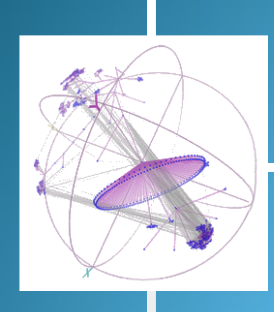
limite	base	read, write, execute?

Évaluation



- Avantages: l'unité d'allocation de mémoire (segment) est :
  - plus petite que le programme entier
  - une entité logique connue par le programmeur
  - les segments peuvent changer de place en mémoire
  - la protection et le partage de segments sont aisés (en principe)
- Désavantage: le problème des partitions dynamiques:
  - La fragmentation externe n'est pas éliminée: trous en mémoire, compression?

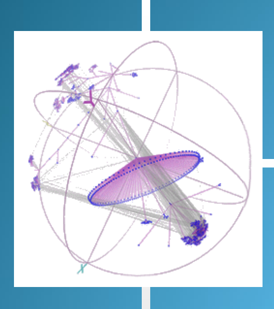
Une autre solution serait d'essayer de simplifier le mécanisme en utilisant des unités d'allocation mémoire de tailles égales : PAGINATION



Pagination

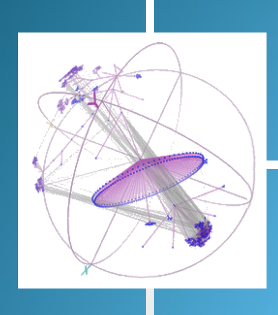
Le problème avec la segmentation est que l’unité d’allocation de mémoire (le segment) est de longueur variable

- La pagination utilise
  - des unités d’allocation de mémoire fixe, éliminant donc ce problème
  - l'allocation non contiguë de blocs de mémoire



Pagination

- Schéma de pagination :
  - La mémoire physique est divisée en blocs de taille fixe (des frames) dont la taille est généralement une puissance de 2 (entre 512 et 8192 octets).
  - La mémoire logique est divisée en blocs de la même taille (des pages)
- Conséquences:
  - Les pages logiques d'un processus peuvent donc être assignés aux cadres disponibles n'importe où en mémoire principale
  - un processus peut être éparpillé n'importe où dans la mémoire physique.
  - la fragmentation externe est éliminée



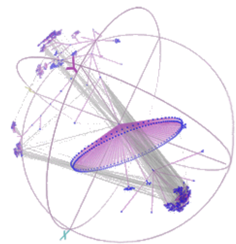
Pagination

Le SE doit en permanence :

- Savoir quelles sont les frames libres.
- Trouver n frames libres si une programme en a besoin.
- Pouvoir gérer les correspondance entre adresses physiques et logiques.

➢ Risque de fragmentation interne à cause des pages.

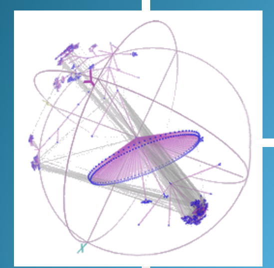
Exemple



Frame number	Main memory	Main memory	Main memory	Main memory
0		A.0	A.0	A.0
1		A.1	A.1	A.1
2		A.2	A.2	A.2
3		A.3	A.3	A.3
4			B.0	B.0
5			B.1	B.1
6			B.2	B.2
7				C.0
8				C.1
9				C.2
10				C.3
11				
12				
13				
14				
	(a) Fifteen Available Pages	(b) Load Process A	(b) Load Process B	(d) Load Process C

- **Supposons que le processus B se termine ou est suspendu**

Exemple



- Nous pouvons maintenant transférer en mémoire un processus D, qui demande 5 cadres, bien qu’il n’y ait pas 5 cadres contigus disponibles
- La fragmentation externe est limitée en cas où le nombre de cadres disponibles n’est pas suffisant pour exécuter un programme en attente
- Seule la dernière page d’un processus peut souffrir de fragmentation interne

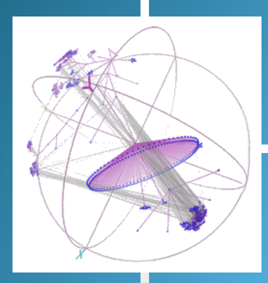
	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(e) Swap out B

	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

(f) Load Process D

Table des pages



- Le SE doit maintenir une table de pages pour chaque processus
- Chaque entrée de la Table de pages est composée de plusieurs champs, notamment :
  1. Le bit de présence.
  2. Le bit de référence (R).
  3. Les bits de protection.
  4. Le bit de modification (M).
  5. Le numéro de cadre contenant la page.
- Une liste de cadres disponibles est également maintenue (free frame list)

0	0
1	1
2	2
3	3

Process A page table

0	—
1	—
2	—

Process B page table

0	7
1	8
2	9
3	10

Process C page table

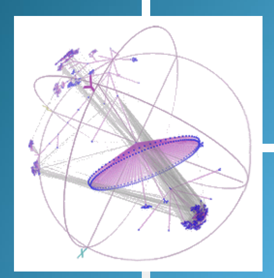
0	4
1	5
2	6
3	11
4	12

Process D page table

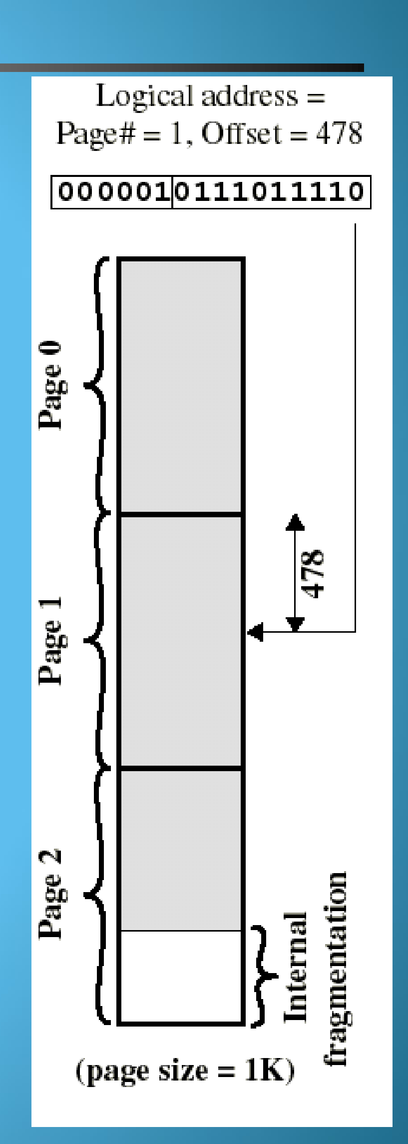
13
14

Free frame list

Adresse logique



- L'adresse logique (n,d) est traduite en adresse physique (k,d) en utilisant n comme index sur la table des pages et en le remplaçant par l'adresse k trouvée
- d ne change pas
- Donc les pages sont invisibles au programmeur,
- La traduction des adresses au moment de l'exécution est facilement réalisable par le matériel: MMU
- Un programme peut être exécuté sur différents matériels employant des dimensions de pages différentes

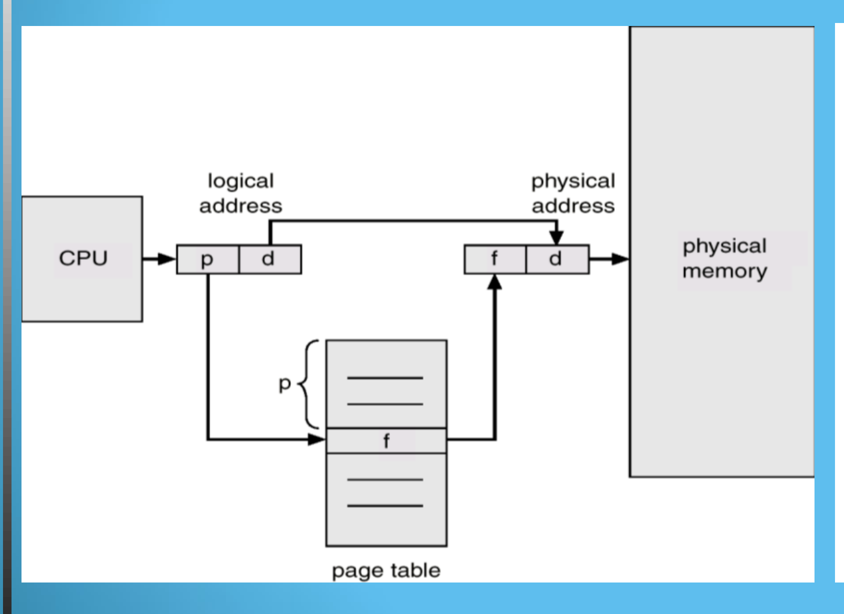


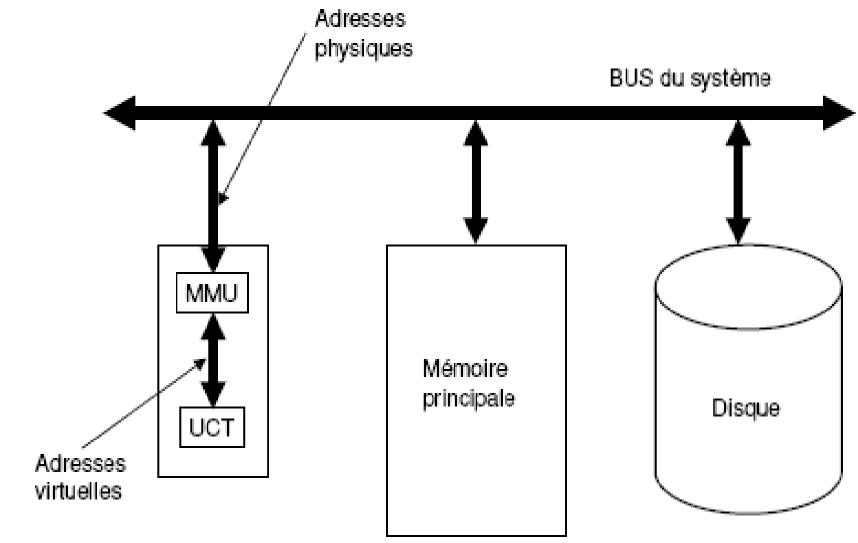


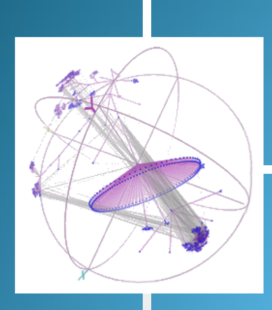
Traduction d'adresses par le MMU

- Traduction d'adresses:

  Tant dans le cas de la segmentation, que dans le cas de la pagination, nous ajoutons donc le décalage à l'adresse du segment ou page.







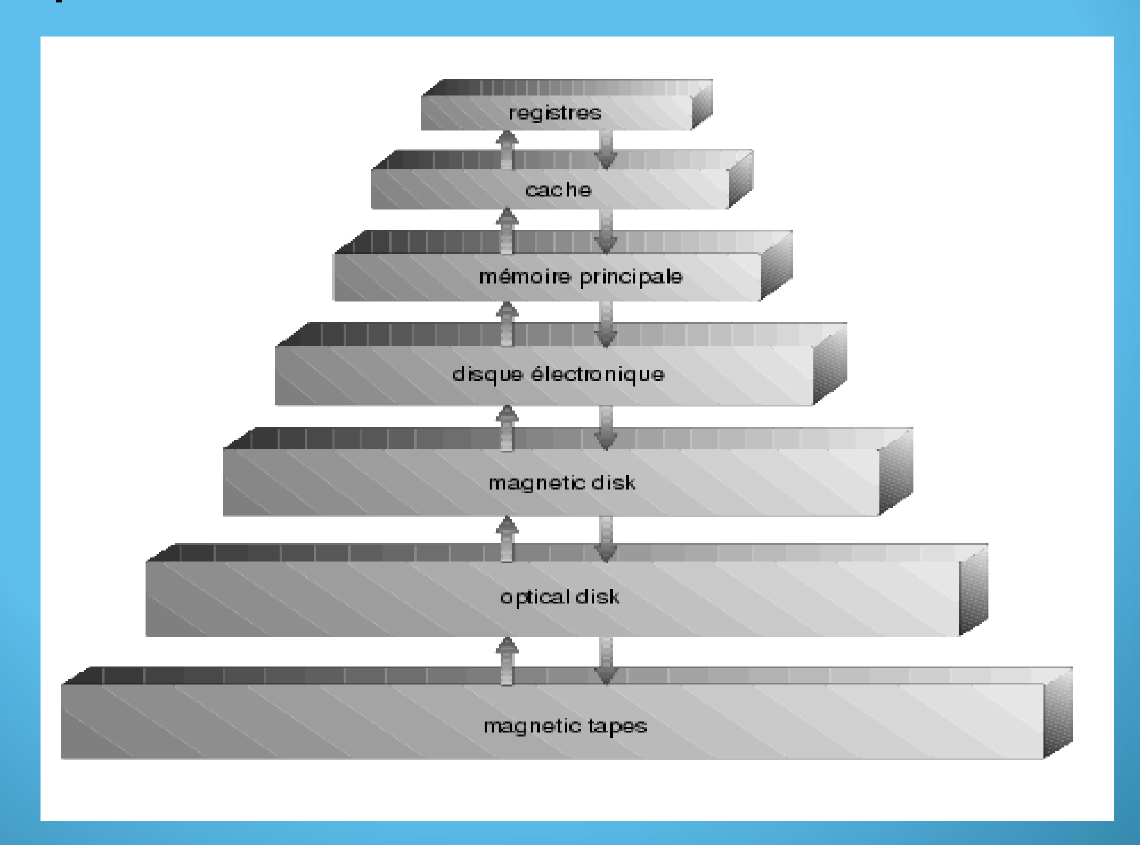
Pagination et segmentation

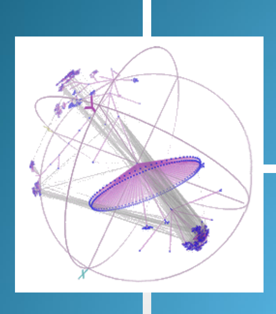
- Techniques combinées utilisées dans de nombreux systèmes modernes :
  - Espace logique divisé en plusieurs segments.
  - Chaque segment est divisé en pages de taille fixe.
  - Si un segment est plus petit qu'une page, il occupe une seule page.
  - Table des pages pour chaque segment.
  - Décalage de segment = numéro de page + décalage de page.
  - Le numéro de frame est combiné avec le décalage de page pour obtenir l'adresse physique.
- Plus de fragmentation externe, mais fragmentation interne et table plus volumineuses.



La mémoire virtuelle

- La mémoire virtuelle est une application du concept de hiérarchie de mémoire

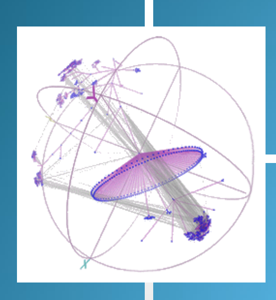




La mémoire virtuelle

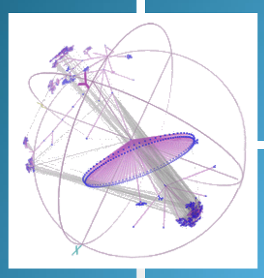
- Séparation de la mémoire logique et de la mémoire physique.
  - Seuls de petites parties des programmes ont besoin d'être en mémoire pour l'exécution.
  - L'adressage logique peut donc être plus étendu que l'espace physique réel.
  - Les pages peuvent donc être ou pas en mémoire physique (mécanismes de swap).

- La mémoire virtuelle peut être implémentée via :
  - Pagination
  - Segmentation



De la pagination et segmentation à la mémoire virtuelle

- Références à la mémoire sont traduites en adresses physiques au moment d’exécution
  - Un processus peut être déplacé à différentes régions de la mémoire, aussi mémoire secondaire!
  - Donc: tous les morceaux d’un processus ne nécessitent pas d’être en mémoire principale durant l’exécution
  - L’exécution peut continuer à condition que la prochaine instruction (ou donnée) soit dans une page se trouvant en mémoire principale
- La somme des mémoires logiques des processus en exécution peut donc excéder la mémoire physique disponible
- Le concept de base de la mémoire virtuelle
  - Une image de tout l’espace d’adressage du processus est gardée en mémoire secondaire (normal. disque) d’où les pages manquantes pourront être prises au besoin
- Mécanisme de va-et-vient ou swapping



Pages en RAM ou sur disque

Page A en RAM **et** sur disque

Page E **seulement** sur disque

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

logical memory

frame

valid–invalid bit

	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

page table

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

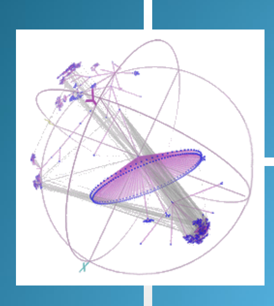
physical memory

A B C D E F





- Ne mettre une page en mémoire que si besoin:
  - Moins d'E/S nécessaires
  - Besoin plus faible en mémoire
  - Plus d'utilisateurs possibles
- Il faut des références sur les pages :
  - Référence invalide $\Rightarrow$ erreur
  - Pas en mémoire $\Rightarrow$ charger la page
  - En mémoire $\Rightarrow$ ok



Bit de validité (présence)

- À chaque page est associé un bit :
  1 $\Rightarrow$ en mémoire, 0 $\Rightarrow$ pas en mémoire
- Initialement tous les bits sont mis à 0.

- Au moment de la conversion des adresses, si le bit est à 0 ; défaut de page(page fault).

- Sauvegarde de l'état du processus.
- Le SE détermine s'il s'agit bien d'une faute de page.
- Le CPU est donné à un autre processus pendant la gestion de la faute.



Plus de pages libres ?

Deux problèmes principaux à gérer. Il faut développer des algorithmes pour :

1. Le remplacement de pages :
   - Trouver une page à sortir de la mémoire : page victime.
   - Évidemment, plusieurs cadres de mémoire ne peuvent pas être `victimisés`: p.ex. cadres contenant le noyau du SE, tampons d ’E/S...
   - La ‘victime’ doit-elle être récrite en mémoire secondaire?
     - Oui, si elle a été modifiée depuis qu’elle a été chargée en mémoire principale
     - sinon, sa copie sur disque est encore fidèle
   - Essayer d'éviter de faire trop de remplacements.
2. L'allocation des frames.

Remplacement de page



frame *bit de validité*

0	i
f	v

Table des pages

4. *Mettre à jour la table*

2. *Invalider la page*

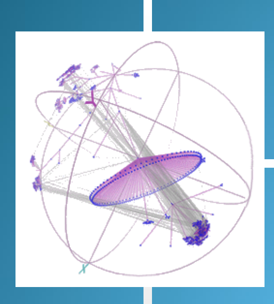
f page

Mémoire physique

1. *Swap out*

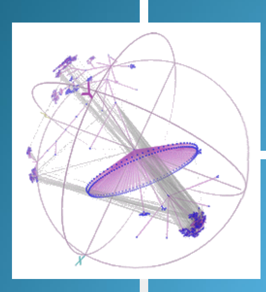
3. *Swap in*

Stockage externe



Algorithmes de remplacement

- Choisir la victime de façon à minimiser le taux de défaut de pages
- Plusieurs méthodes pour choisir qui sortir de la mémoire :
  - First-in-First-Out
  - Optimal
  - Least Recently Used
  - Deuxième chance : horloge (clock)
  - …
- On cherche un algorithme qui fait peu de défauts de pages.
- Évaluation de l'algorithme sur une suite de demandes.



Algorithme FIFO et LRU

FIFO

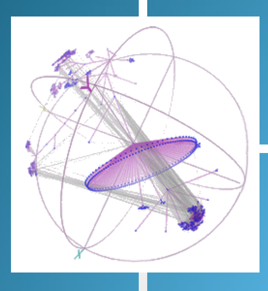
- Pour chaque page on connaît son heure d'arrivée en mémoire.
- Quand une page doit être remplacée, on choisit la plus vieille.
  - Problème :Les premières pages amenées en mémoire sont souvent utiles pendant toute l’exécution d’un processus!
  - variables globales, programme principal, etc.

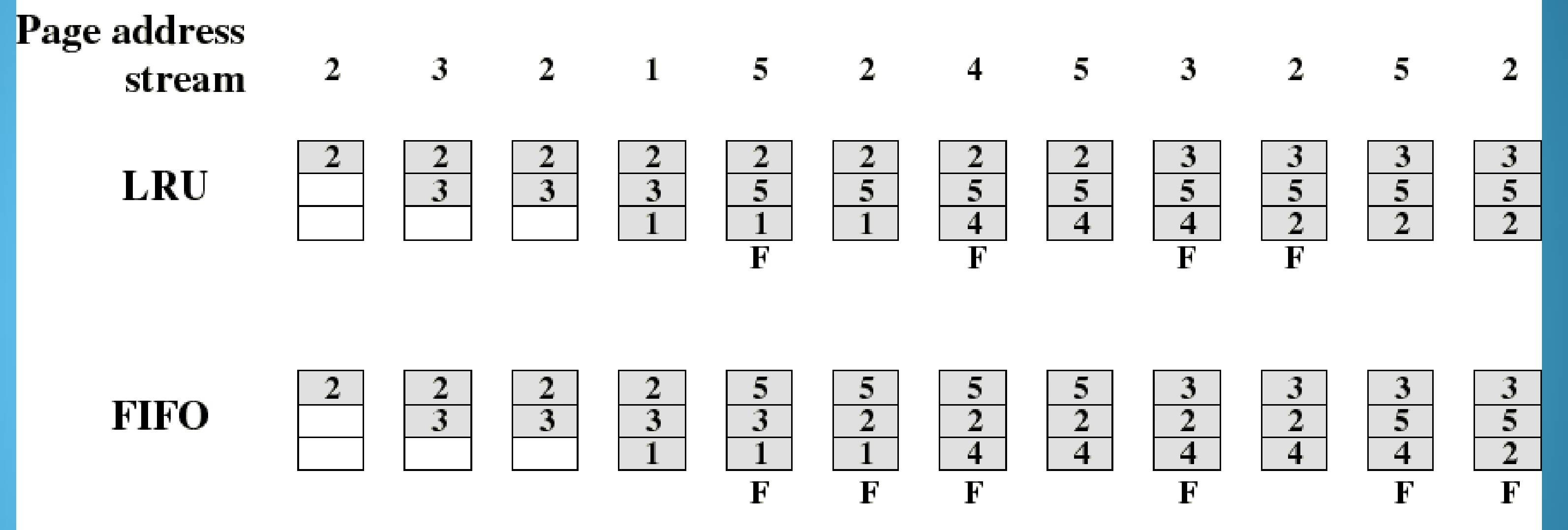
Least-recently-used

La moins récemment utilisée

- Remplacer la page qui n'a pas été utilisée depuis le plus longtemps
- Associer à chaque page son dernier instant d'utilisation.

Comparaison de FIFO avec LRU

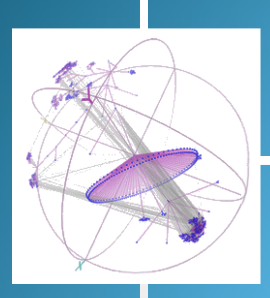




Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F

- Contrairement à FIFO, LRU reconnaît que les pages 2 and 5 sont utilisées récemment
- La performance de FIFO est moins bonne:

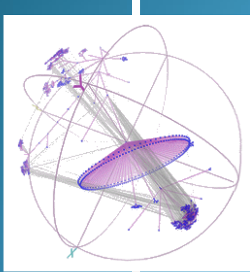
  LRU: 3+4=7, FIFO: 3+6=9



Comparaison OPT-LRU

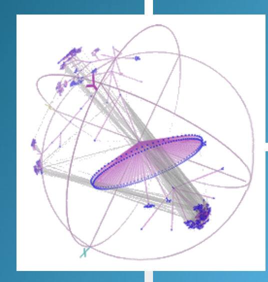
OPT: 3+3=6, LRU 3+4=7 .

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
					F		F			F		
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		



L’algorithme de l’horloge

- Semblable à FIFO, mais les cadres qui viennent d’être utilisés (bit=1) ne sont pas remplacées (deuxième chance)
- Les cadres forment conceptuellement un tampon circulaire
- Lorsqu’une page est chargée dans un cadre, un pointeur pointe sur le prochain cadre du tampon
- Pour chaque cadre du tampon, un bit “utilisé” est mis à 1 (par le matériel) lorsque:
  - ✓ une page y est nouvellement chargée
  - ✓ sa page est utilisée
- Le prochain cadre du tampon à être remplacé sera le premier rencontré qui a son bit “utilisé” = 0.
- Durant cette recherche, tout bit “utilisé” = 1 rencontré sera mis à 0



Algorithme de l'horloge Remplacement global

n
0
page 9 use = 1
page 19 use = 1
1
page 1 use = 0
next frame pointer
page 45 use = 1
2
page 222 use = 0
8
page 191 use = 1
3
page 33 use = 1
7
page 556 use = 0
4
page 67 use = 1
page 13 use = 0
6
5

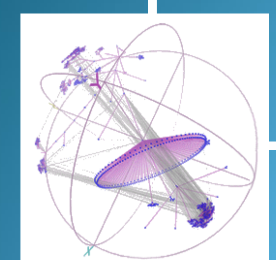
(a) State of buffer just prior to a page replacement

n
0
page 9 use = 1
page 19 use = 1
1
page 1 use = 0
page 45 use = 0
2
page 222 use = 0
8
page 191 use = 0
3
page 33 use = 1
7
page 727 use = 1
4
page 67 use = 1
page 13 use = 0
6
5

(b) State of buffer just after the next page replacement

La page 727 est chargée dans le cadre 4.
La prochaine victime est 5, puis 8.

Algorithme de l'horloge Remplacement local

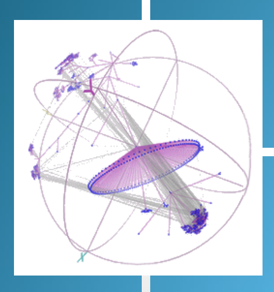


Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F
CLOCK	2*	2*	2*	2*	5*	5*	5*	5*	3*	3*	3*	3*
		3*	3*	3*	3	2*	2*	2*	2	2*	2	2*
				1*	1	1	4*	4*	4	4	5*	5*
					F	F	F		F		F	

- Astérisque indique que le bit utilisé est 1
- L'horloge protège du remplacement les pages récemment utilisées en mettant à 1 le bit "utilisé" à chaque référence

OPT: 3+3=6, LRU: 3+4=7, Horloge: 3+5=8 , FIFO: 3+6=9

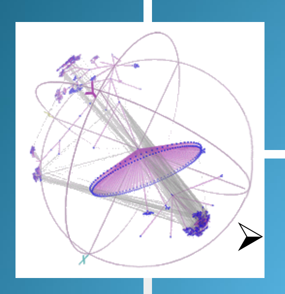
Anomalie de Belady



1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

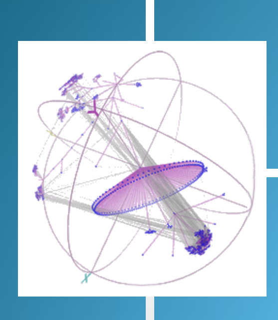
- 9 fautes de page.
- En général, plus il y a de frames, moins il y a de fautes.
  - *Anomalie de Belady*.
- 9 fautes de page.

1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3



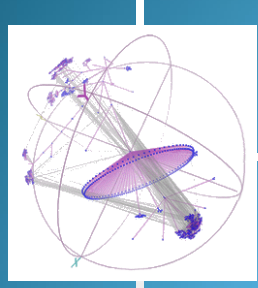
Comparaison des algorithmes

- Les simulations montrent que l'horloge est presque aussi performant que LRU
- variantes de l'horloge ont été implantées dans des systèmes réels
- Lorsque les pages candidates au remplacement sont locales au processus souffrant du défaut de page et que le nombre de cadres alloué est fixe, les expériences montrent que:
  - ✓ Si peu (6 à 8) de cadres sont alloués, le nombre de défaut de pages produit par FIFO est presque double de celui produit par LRU, et celui de CLOCK est entre les deux
  - ✓ Ce facteur s'approche de 1 lorsque plusieurs (plus de 12) cadres sont alloués.
- Cependant le cas réel est de milliers et millions de pages et cadres, donc la différence n'est pas trop importante en pratique...
- On peut tranquillement utiliser LRU



Algorithmes compteurs

- Garder un compteur pour les références à chaque page
- LFU: Least Frequently Used: remplacer la pages avec le plus petit compteur
- MFU: Most Frequently Used: remplacer les pages bien usées pour donner une chance aux nouvelles
- Ces algorithmes sont d'implantation couteuse et ne sont pas très utilisés

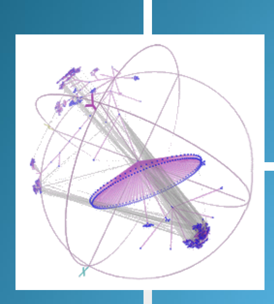


Thrashing

- S'il y a trop de défauts de page, un processus peut passer plus de temps en attente de pagination qu'en exécution : trashing.
  - Performance très amoindrie du système.

Exemple :

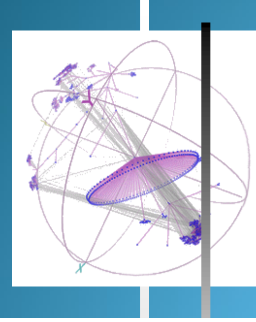
- Utilisation faible du CPU : de nouveaux processus sont admis. Si un processus a besoin de plus de frames, il va faire des fautes et prendre des frames aux autres processus. Ceux-ci vont à leur tour prendre des frames aux autres, etc.
- Ces processus doivent attendre que les pages soient chargés et vont donc être en attente, donc l'utilisation du CPU diminue.

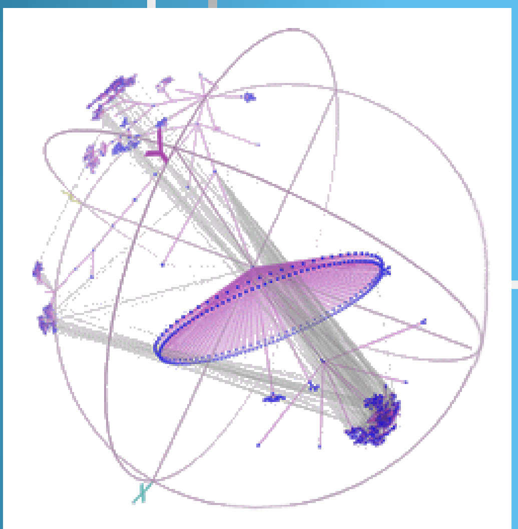


Thrashing (2)

Comment gérer le trashing ?

- Algorithme de remplacement local : un processus qui fait du trashing ne peut pas prendre de frames aux autres processus.
- Fournir à un processus autant de frames qu'il en a besoin. Utiliser par exemple le *Working Set Model.*
  - Working set : ensemble de travail = ensemble des pages qu'un processus utilise.
  - Si l'ensemble de travail est entièrement en mémoire, le processus ne fera pas de faute.
  - Le système de pagination doit s'assurer qu'il l'ensemble de travail est en mémoire avant que le processus ne puisse avoir accès au processeur.

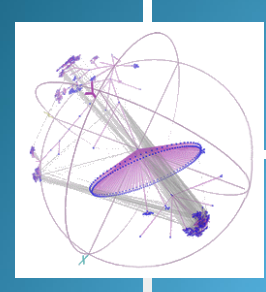




Chapitre 4
Le système de Gestion de Fichiers

1. **Systèmes d'entrée/sortie**
2. **Systèmes de fichiers**
3. **Structure de mémoire de masse (disques)**

Systèmes d’entrée/sortie



Concepts importants :

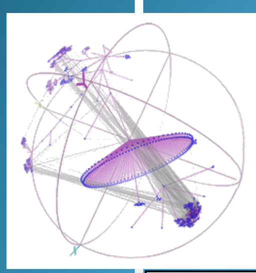
- Matériel E/S
- Communication entre UCT et contrôleurs périphériques
- DMA
- Pilotes et contrôleurs de périfs
- Sous-système du noyau pour E/S
  - Tamponnage, cache, spoule



Contrôleurs de périphériques

- On se rappel: Les périphériques d’E/S ont typiquement une composante mécanique et une composante électronique
  - La partie électronique est le contrôleur

Monitor
Monitor
Keyboard
Floppy disk drive
Hard disk drive
CPU
Memory
Video controller
Keyboard controller
Floppy disk controller
Hard disk controller
Bus
Bus



Revue des disques magnétiques

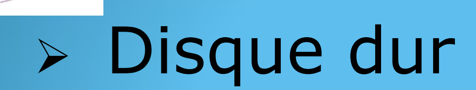
Read/write head (1 per surface)

Surface 7
Surface 6
Surface 5
Surface 4
Surface 3
Surface 2
Surface 1
Surface 0

Direction of arm motion

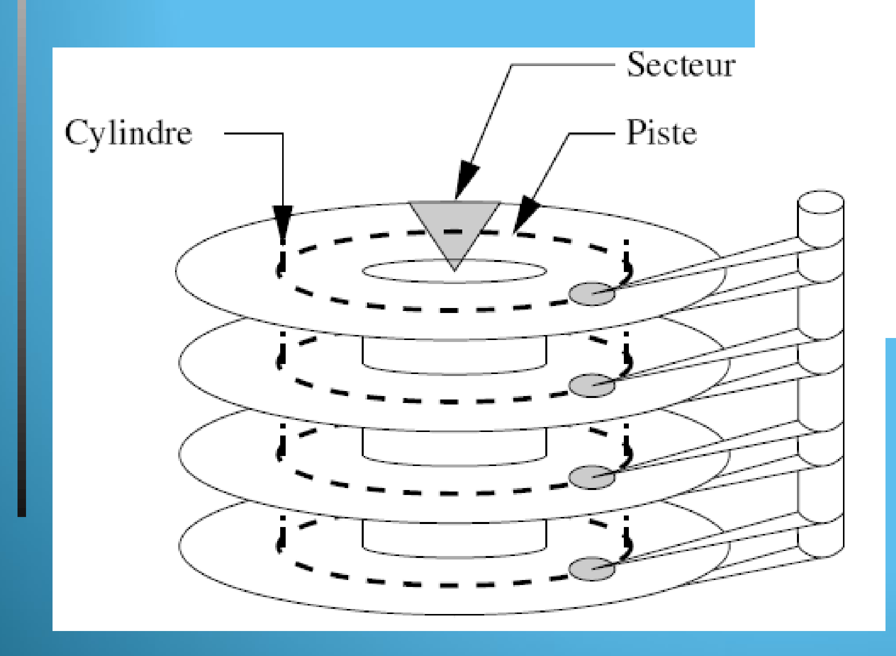
- Les disques sont organisés en cylindres, pistes et secteurs

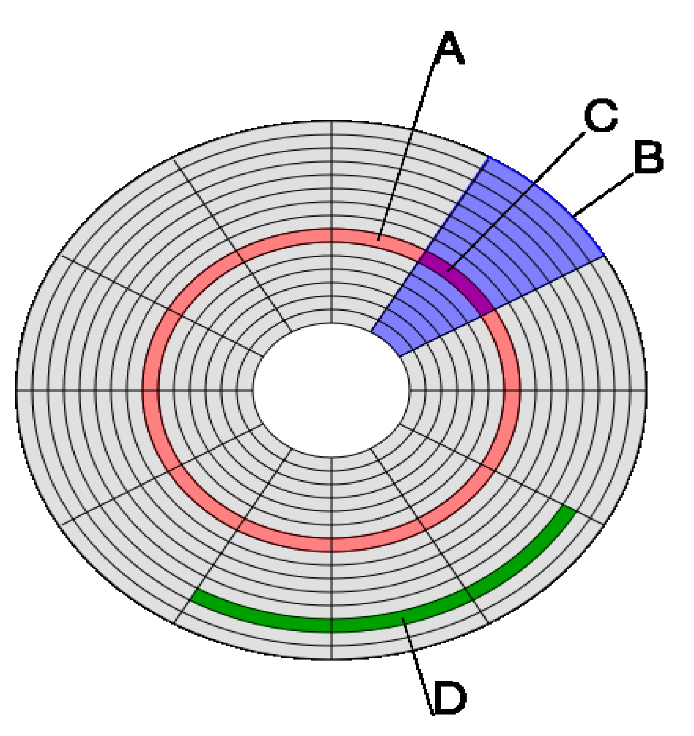
Support physique de codage de l'information

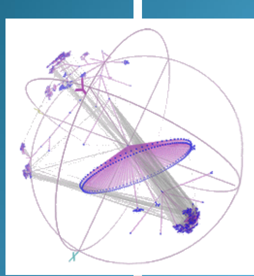


- Disque dur

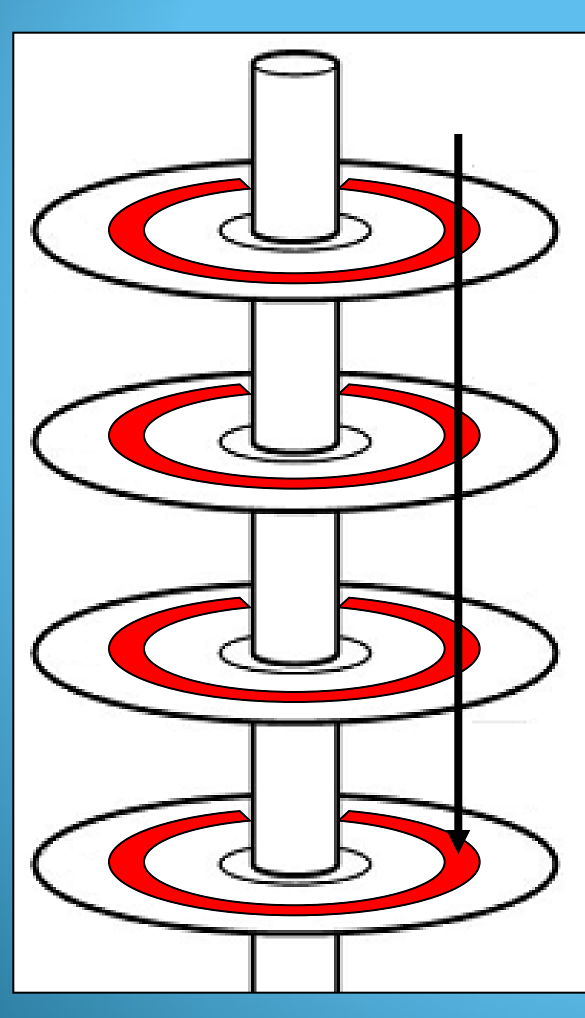
(A) Piste
(B) Secteur géométrique
(C) secteur d'une piste
(D) cluster



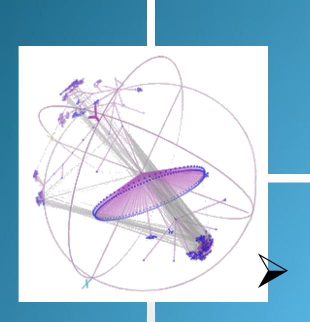




Revue des disques magnétiques

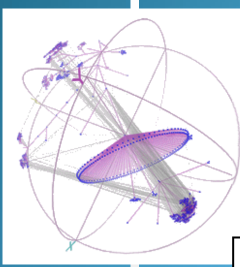


- Toutes les pistes pour une position donnée du bras forment un cylindre.
  - Donc le nombre de cylindre est égale au nombre de piste par côté de plateau
  - La location sur un disque est spécifié par (cylindre, tête, secteur)



Disques magnétiques

- Sur les disques plus vieux, le nombre de secteurs par piste étaient constant pour tout les cylindres
  - Ceci gaspille de l'espace de stockage potentiel sur les cylindres du disque
  - Les disques modernes réduisent le nombre de secteurs par piste vers l'intérieur du disque. Certains par paliers (plusieurs pistes) certains de façon linéaire
  - Souvent, les contrôleurs modernes attachés aux disques présentent une géométrie virtuelle au système d'exploitation, mais ont un arrangement physique beaucoup plus compliqué...

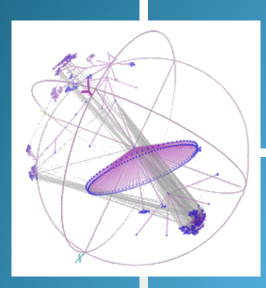


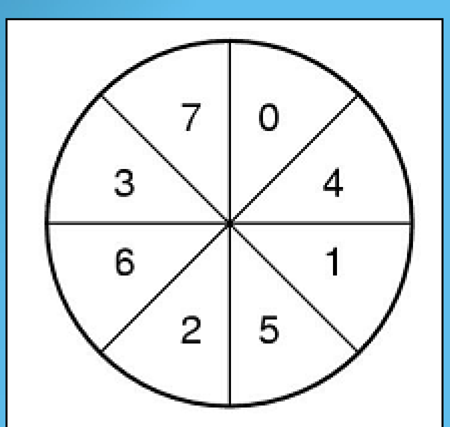
Disques magnétiques

Organisation virtuelle

Géométrie Physique

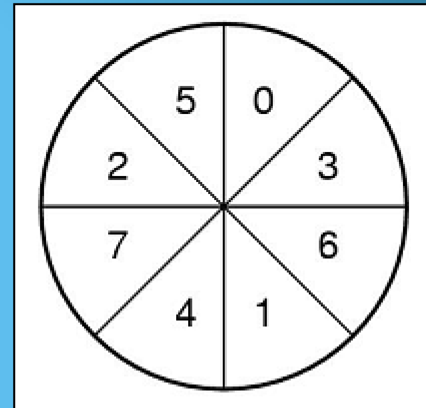




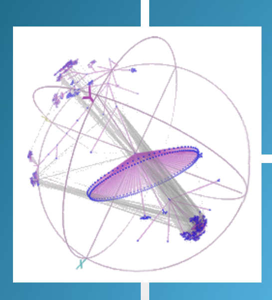


Entrelacement simple

Entrelacement double



- L’entrelacement permet au disque de tourner et de passer n secteurs et de prendre le prochain secteur virtuellement contiguë pendant que les données sont transférés du contrôleur vers la mémoire



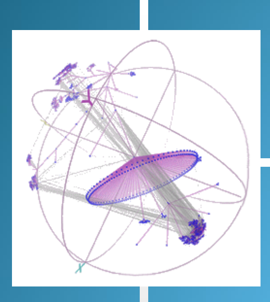
Les Périphériques

- Périphériques blocs ou caractères
  - Périphériques blocs: disques, rubans…
    - Commandes: read, write, seek
    - Accès brut (raw) ou à travers système fichiers
    - Accès représenté en mémoire (memory-mapped)
      - Semblable au concept de mémoire virtuelle ou cache:
        - » une certaine partie du contenu du périphérique est stocké en mémoire principale(cache), donc quand un programme fait une lecture de disque, ceci pourrait être une lecture de mémoire principale
  - Périphériques par caractère (écran)
  - Get, put traitent des caractères
  - Librairies au dessus peuvent permettre édition de lignes, etc.

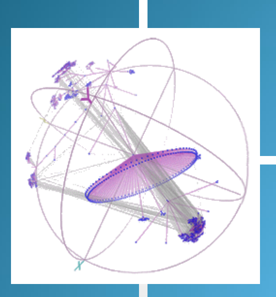
Pilotes de périphériques

- Pilotes de périphériques
  - Chaque périphérique d’E/S attaché à l’ordinateur requiert du code spécifique pour faire l’interface entre le matériel et le SE. Ce code s’appel pilote de périphérique
    - Ceci est parce qu’au niveau du matériel, les périphériques sont radicalement différents les uns des autres
    - Parfois un pilote va prendre soins d’une classe de périphériques qui sont proche ex.: un nombre de souris
  - Les pilotes de périphériques sont normalement produit par le manufacturier du périphérique pour les SEs populaires





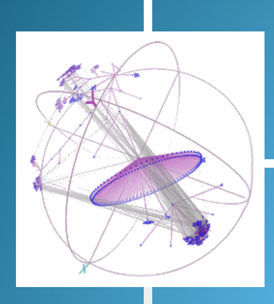
- Pilotes de périphériques
  - Typiquement les pilotes sont dans le noyau pour qu'ils puissent avoir accès au registres de contrôle du périphérique
    - Ce n'est pas un requis. Vous pourriez avoir un pilote dans l'espace utilisateur et faire des appels de systèmes pour communiquer avec les registres. Par contre la pratique courante est d'avoir les pilotes dans le noyau.
  - Étant donné que c'est la méthode habituel d'implémenter les pilotes, l'architecture normale est de mettre les pilotes 'en bas' du SE





Pilotes de périphériques

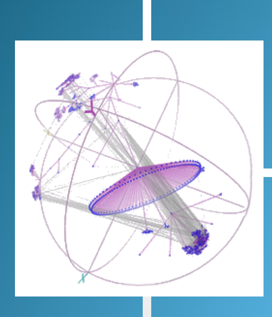
- Que font les pilotes de périphériques?
  - Ils acceptent les commandes abstraites de lecture/écriture de la couche supérieure
  - Fonctions assorties:
    - Initialise le périphérique
    - Gère la puissance – Arrête un disque de tourner, ferme un écran, ferme une caméra, etc.





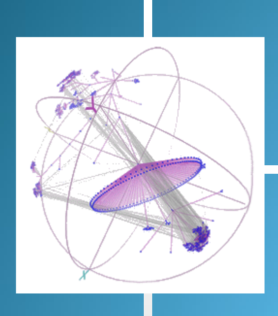
Pilotes de périphériques

- Qu’est-ce que un pilote fait sur une lecture/écriture?
  - Vérifie les paramètres d’entrée & retourne les erreurs
  - Converti les commandes ***abstraites*** (lit le secteur) en commandes ***physiques*** (tête, traque, secteur, et cylindre)
  - Met les demandes dans une queue si le périphérique est occupé
  - Amène le périphérique en état de fonctionnement si requis
    - Monter la vitesse du moteur, température, etc.
  - Contrôle le périphérique en envoyant des commandes par les registres de contrôle



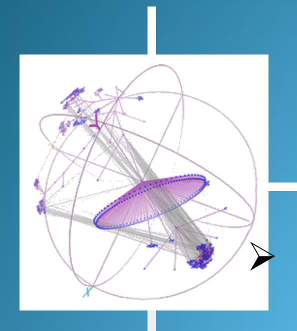
Contrôleurs de périphériques

- Sur un PC, le contrôleur de périphérique est habituellement sur un circuit imprimé
  - Il peut être intégré sur la carte mère
- Le job du contrôleur est de convertir un flot de série de bits en octets ou en blocs d'octets et de faire les conversions et corrections
  - En fin de compte touts les périphériques traitent des bits. C'est le contrôleur qui groupe ou dégroupe ces bits



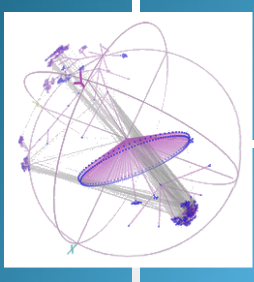
Registres de contrôle

- Chaque contrôleur possède quelques registres qui servent à la communication avec le processeur
  - Le système d'exploitation peut commander les périphériques de fournir ou accepter des données, de s'éteindre, etc.
  - Le SE peut aussi lire certains registres pour déterminer l'état du périphérique
- Les contrôleurs ont *typiquement* des tampons de données que le SE peut lire et écrire
  - Important pour ne pas perdre de données
  - Parfois utilisé comme 'partie' du périphérique, ie: RAM vidéo dans laquelle les programmes ou le SE peuvent écrire



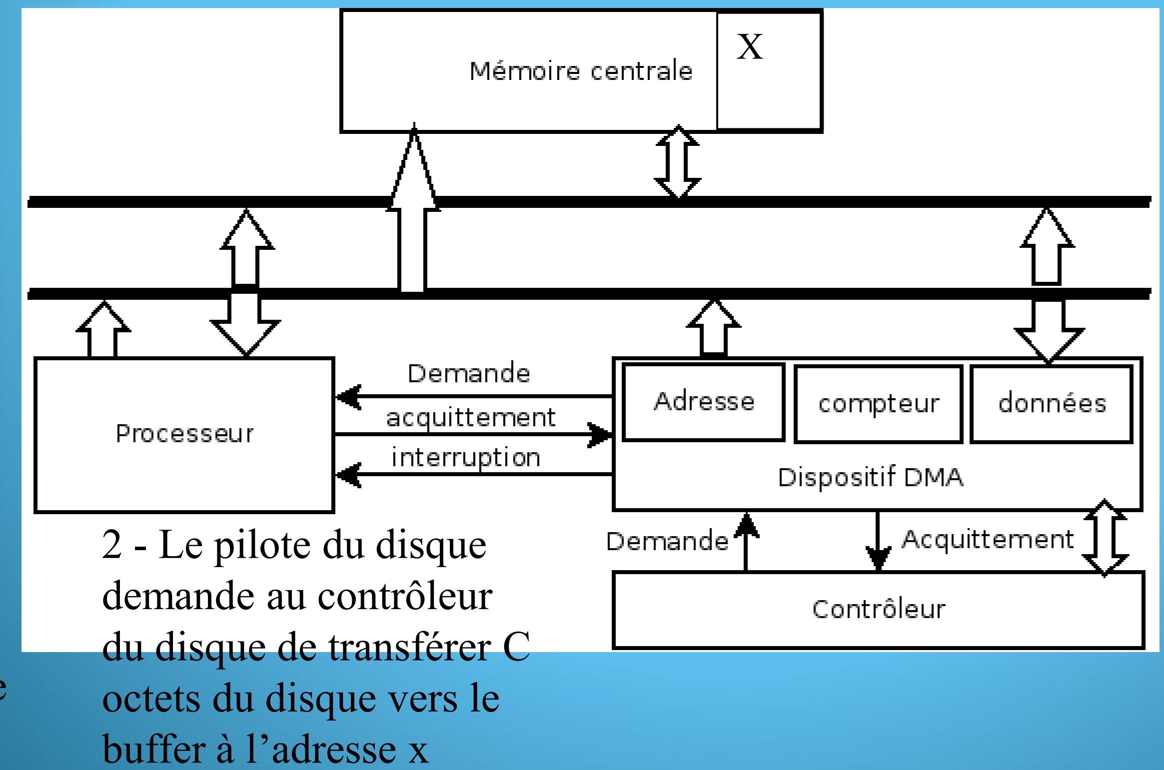
Communication entre UCT et contrôleurs périphériques

- Deux techniques de base:
  - UCT et contrôleurs communiquent directement par des registres
  - UCT et contrôleurs communiquent par des zones de mémoire centrale
  - Combinaisons de ces deux techniques



Accès direct en mémoire (DMA)

- Dans les systèmes sans DMA, l'UCT est impliquée dans le transfert de chaque octet
- DMA est utile pour exclure l'implication de l'UCT surtout pour des E/S volumineuses
- Demande un contrôleur spécial a accès direct à la mémoire centrale MMU (Memory Management Unit)



1- CPU demande au pilote du périphérique (HD)

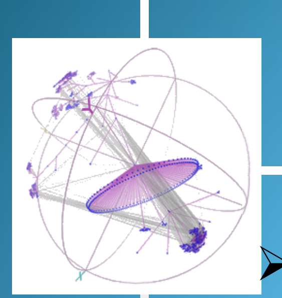
2 - Le pilote du disque demande au contrôleur du disque de transférer C octets du disque vers le buffer à l'adresse x

3 - Le contrôleur du disque initie le transfert DMA

4 - Le contrôleur du disque envoie chaque octet au contrôleur du DMA

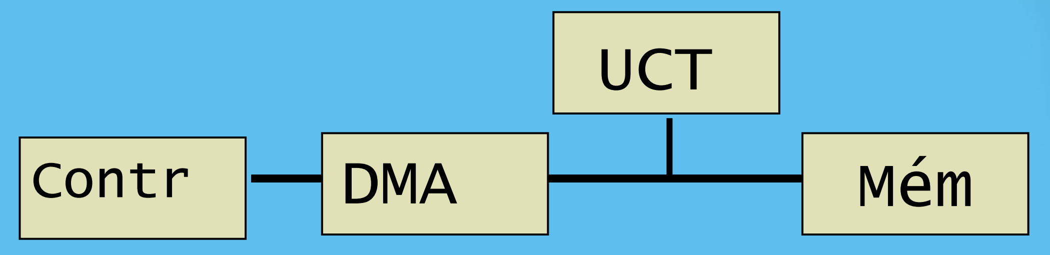
5 - Le contrôleur DMA transfert les octets au buffer x en augmentant l'adresse x et décrémentant le compteur c

6 - Lorsque C=0 DMA envoie une interruption pour signaler la fin du transfert



Vol de cycles

- Le DMA ralentit encore le traitement d'UCT car quand le DMA utilise le bus mémoire, l'UCT ne peut pas s'en servir



- Mais beaucoup moins que sans DMA, quand l'UCT doit s'occuper de gérer le transfert

Mémoire <-> Périphérique





Sous-système E/S du noyau

- Fonctionnalités:
  - Mise en tampon
  - Mise en cache
  - Mise en attente et réservation de périphérique spoule
  - Gestion des erreurs
  - Ordonnancement E/S
    - Optimiser l’ordre dans lequel les E/S sont exécutées

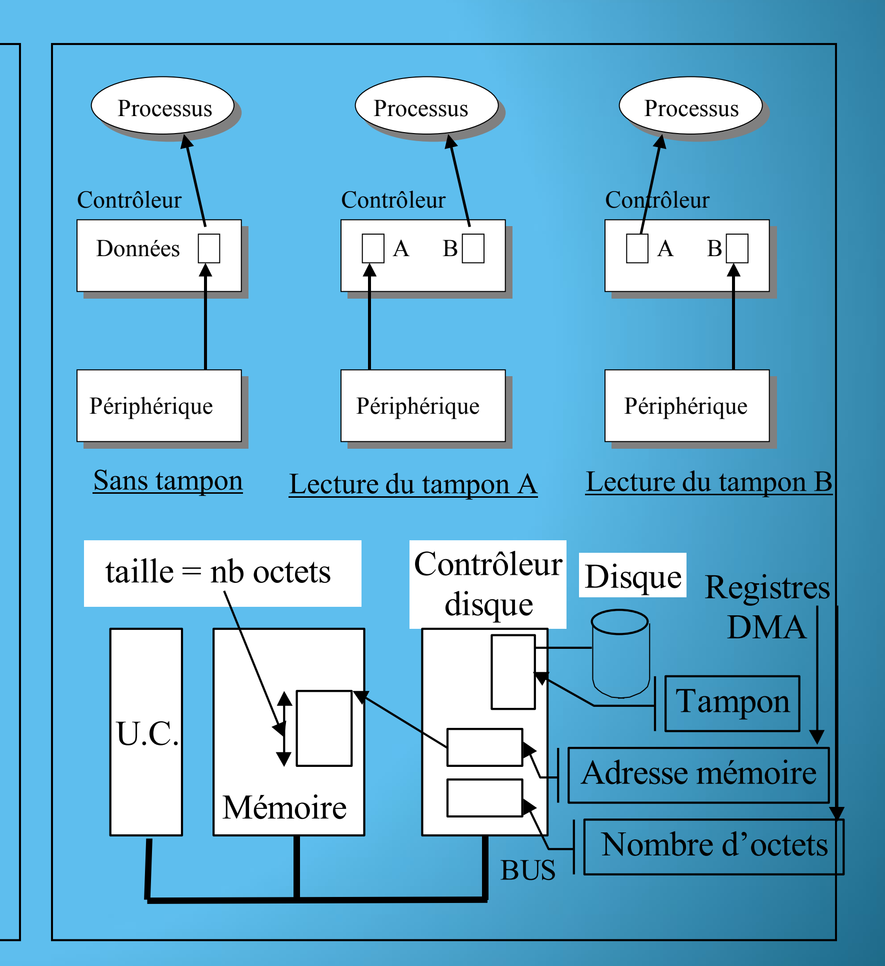


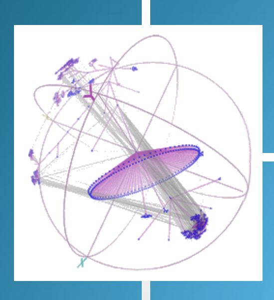
Mise en tampon

- Double tamponnage:
  - P.ex. en sortie: un processus écrit le prochain enregistrement sur un tampon en mémoire tant que l’enregistrement précédent est en train d’être écrit
  - Permet superposition traitement E/S



- Principes.
  - Simultanéité des opérations d'entrées et de sorties avec les opérations de calcul.
  - Le contrôleur de périphérique inclue plusieurs registres de données.
  - Pendant que l'UCT accède à un registre, le contôleur peut accéder à un autre registre.





Mise en cache

- Quelques éléments couramment utilisés d’une mémoire secondaire sont gardés en mémoire centrale

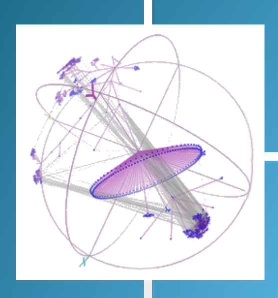
- Donc quand un processus exécute une E/S, celle-ci pourrait ne pas être une E/S réelle:
  - Elle pourrait être un transfert en mémoire, une simple mise à jour d’un pointeur, etc.



Mise en attente et réservation de périphérique: spool

- Spool ou Spooling est un mécanisme par lequel des travaux à faire sont stockés dans un fichier, pour être ordonnancés plus tard

- Pour optimiser l'utilisation des périphériques lents, le SE pourrait diriger à un stockage temporaire les données destinés au périphérique (ou provenant d'elle)
  - P.ex. chaque fois qu'un programmeur fait une impression, les données pourraient au lieu être envoyées à un disque, pour être imprimées dans leur ordre de priorité
  - Aussi les données en provenance d'un lecteur optique pourraient être stockées pour traitement plus tard



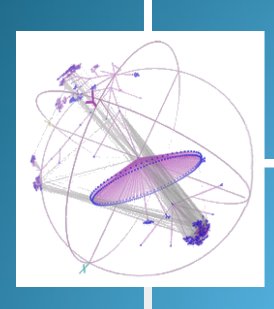
Gestion des erreurs

- Exemples d’erreurs à être traités par le SE:
  - Erreurs de lecture/écriture, protection, périph non-disponible
- Les erreurs retournent un code ‘raison’
- Traitement différent dans les différents cas…



Gestion de requêtes E/S

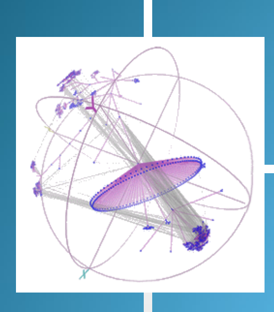
- P. ex. lecture d’un fichier de disque
  - Déterminer où se trouve le fichier
  - Traduire le nom du fichier en nom de périphérique et location dans périphérique
  - Lire physiquement le fichier dans le tampon
  - Rendre les données disponibles au processus
  - Retourner au processus



Systèmes de fichiers

Concepts importants :

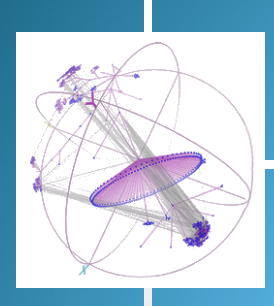
- Systèmes fichiers
- Méthodes d’accès
- Structures Répertoires
- Structures de systèmes fichiers
- Méthodes d’allocation
- Gestion de l’espace libre
- Implémentation de répertoires



Le concept de fichier

- Un fichier est un ensemble nommé d'informations reliées qui est stocké sur tout type de système de stockage
  - Programmes, données, etc.

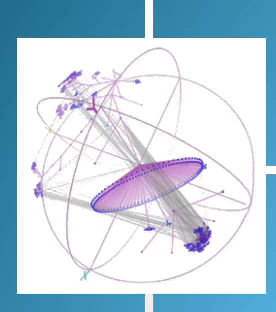
- Le système gère la correspondance entre fichiers et périphériques de stockage matériel (disques, bandes magnétiques, disquettes, CDs, DVDs, etc.)



Attributs d'un fichier

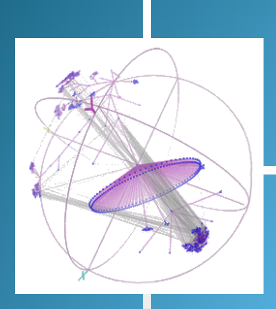
- Nom : sous forme lisible.
- Identifiant : généralement un entier.
- Type : exécutable, image, etc.
- Emplacement : pointeur vers le périphérique et la position sur celui-ci.
- Taille : réelle et éventuellement maximum possible.
- Protections : lecture, écriture et exécution.
- Heure, date et utilisateur : pour la création, la dernière modification et dernière utilisation.

- Ces informations sont stockées dans la structure en répertoires du disque.



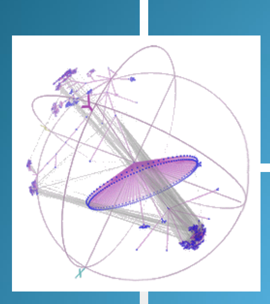
Opérations sur les fichiers

- Création
  - Trouver l'espace dans le système de fichier
  - Ajouter les entrées dans les répertoires
- Écriture : à la position courante
- Lecture : à la position courante
- Repositionnement dans le fichier : modifier la position courante
- Suppression : supprimer l'entrée et libérer l'espace disque
- Vider : mettre la taille à 0 (libération de l'espace) dans modifier les autres attributs



Implémentation

- Les fichiers doivent être ouvert avant d'effectuer lectures et écritures et fermés ensuite.
  - Une seule recherche du fichier dans le répertoire
- Open(filename) – recherche le fichier dans le répertoire et copie l'entrée en mémoire dans une table des fichiers ouverts. Retourne un pointeur vers cette entrée.
- Unix: write(fd, data, bytes), read(fd, data, bytes)
  - fd – index dans la table des fichiers ouverts
- Close (F) – déplace le contenu de l'entrée F dans la structure du répertoire et supprime F de la table des fichiers ouverts



Implémentation

- Dans un système multi-utilisateurs, plusieurs processus peuvent ouvrir simultanément le même fichier.

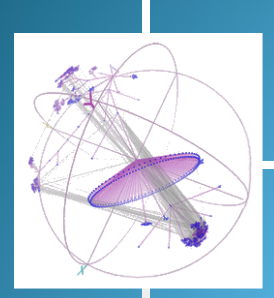
Deux tables pour les fichiers ouverts :

- Table globale : informations sur les fichiers (nom, emplacement, dates d'accès, taille, etc.)
  - Compteur du nombre d'ouverture pour chaque fichier : chaque fermeture décrémente ce compteur et l'entrée est supprimée quand il atteint 0.
- Table locale au processus
  - Chaque entrée contient la position dans le fichier, les droits d'accès et un pointeur vers une entrée dans la table globale.

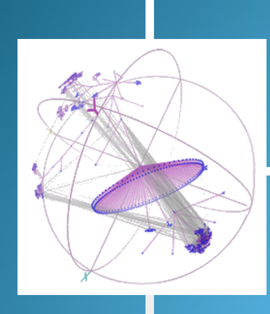


Structure en répertoires

- Les disques peuvent être partagés en partitions.
- Une partition peut s'étendre sur plusieurs disques
- Chaque partition à un système de fichiers propre :
  - Répertoires
  - Fichiers

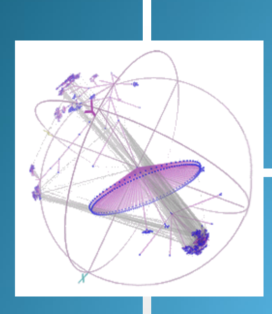


Organisation typique



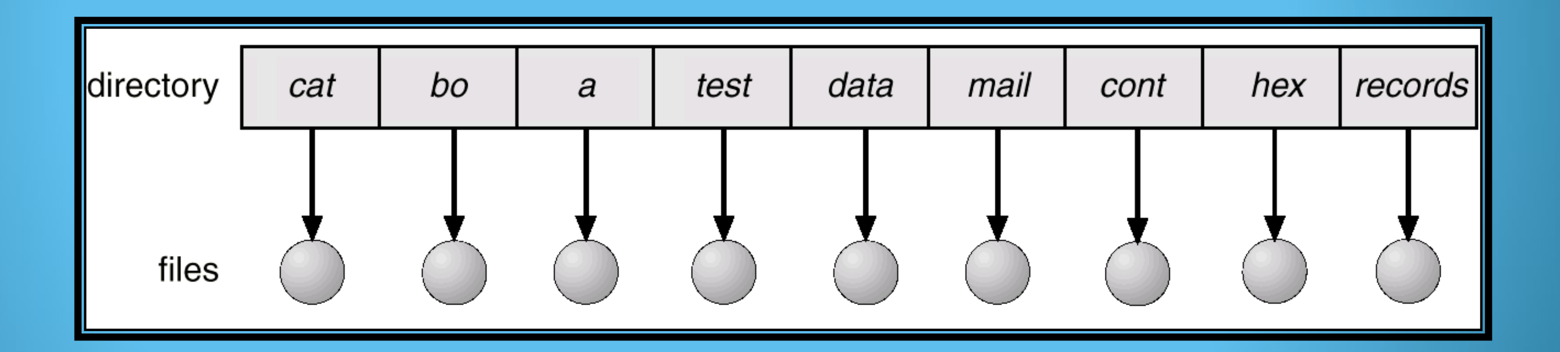
Agir sur les répertoires

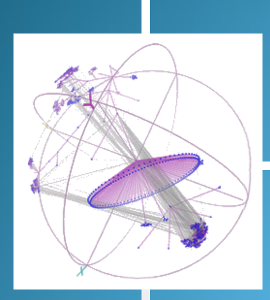
- Chercher un fichier
- Créer un fichier
- Supprimer un fichier
- Lister un répertoire
- Renommer un répertoire.
- Traverser le système de fichiers (visiter l'ensemble de la structure)



Répertoires à un niveau

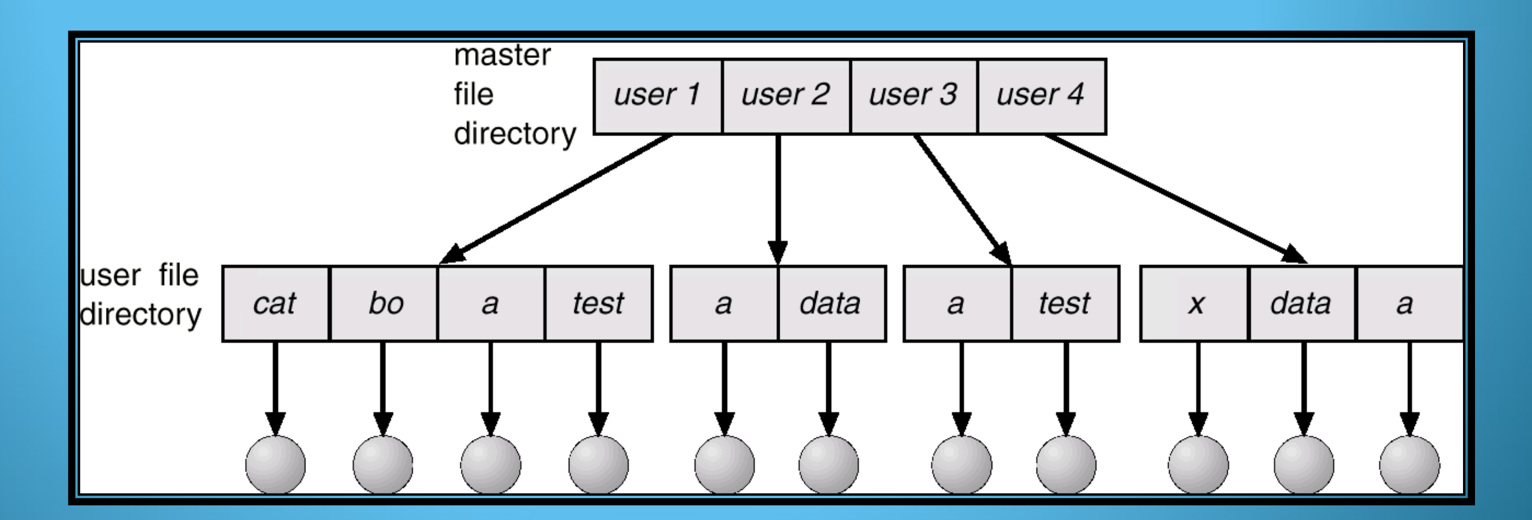
- Un seul répertoire pour tous les utilisateurs.
- Tous les fichiers doivent avoir un nom unique (dur à gérer s'il y a beaucoup d'utilisateurs ou de fichiers)

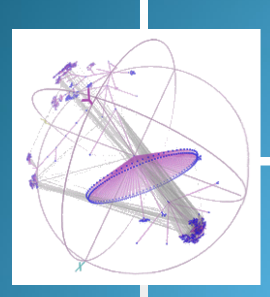




Répertoires à deux niveaux

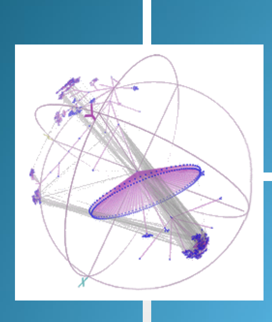
- Un répertoire par utilisateur
  - Noms de fichiers identiques pour des utilisateurs différents
  - Recherche efficace
- Chaque fichier est identifié par nom de l'utilisateur + nom du fichier





Répertoires arborescents

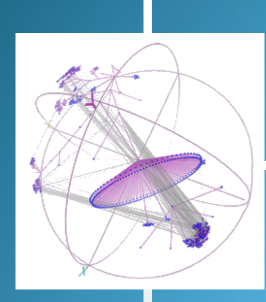
root
spell bin programs
stat mail dist
find count hex reorder
p e mail
prog copy prt exp
reorder list find
hex count
list obj spell
all
last first



Répertoires arborescents

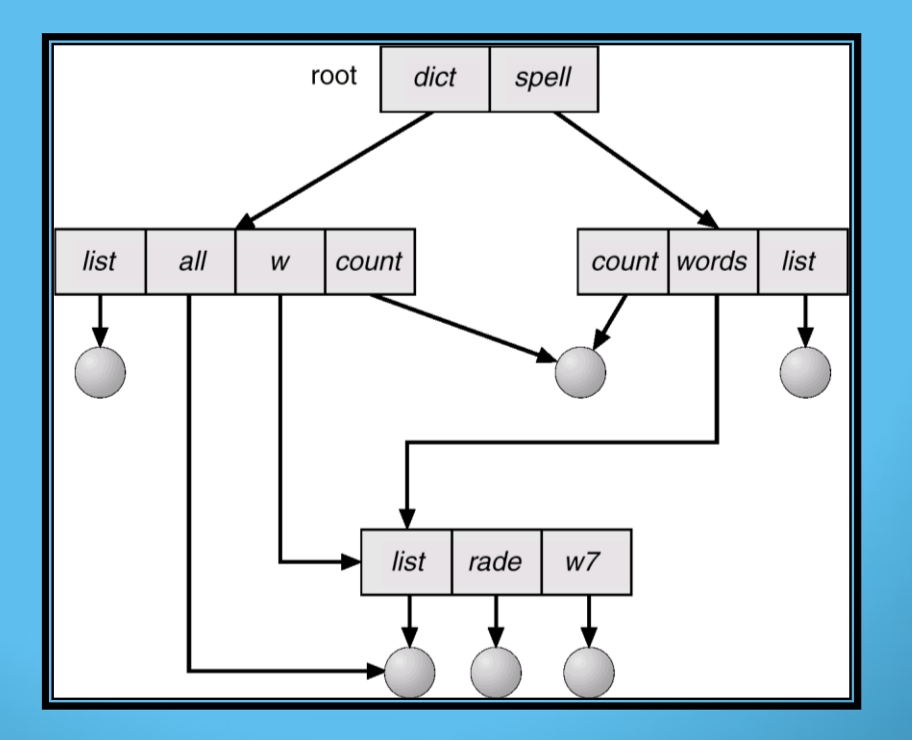
- Possibilité de créer des sous-repertoires
- Un répertoire contient des fichiers et des sous-repertoires

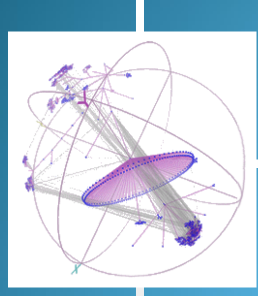
- Chaque fichier à un unique chemin (path name)
- Références absolues ou relatives :
  - À partir de la racine
  - Relatives au répertoire courant



Structure en graphe sans cycle

- Permet le partage de fichiers et de répertoires
- Un seul fichier existe réellement en cas de partage => les modifications faites par un utilisateur sont visibles par d'autres

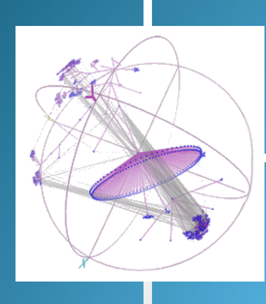




Montage du système de fichiers

- Un système de fichiers doit être monté avant d'être accessible.
- Un système non monté est monté dans la structure à un point de montage.

- UNIX: la commande ‘mount’ permet d'attacher le système de fichiers d'un périphérique dans l'arborescence.
- Windows: les périphériques et partitions ont une lettre pour les identifier.



(a) Partition existante.

(b) Partition non montée.

/
users
bill
fred
help
sue
jane
prog
doc

(a) (b)



Partage à distance

- Dans un système distribué, les fichiers peuvent être partagé via un réseau :
  - Transfert manuel (FTP)
  - Système de fichiers distribué (DFS)
  - World Wide Web (WWW)

- Le modèle client-serveur
  - Le serveur contient les fichiers
  - Le client veut y accéder
  - Ils communiquent en utilisant un protocole d'échange



Protection

- Protection des fichiers contre les accès non désirés
- Le possesseur doit pouvoir contrôler :
  - Ce qui peut être fait
  - Par qui
- Types d'accès pouvant être contrôlés
  - Lecture
  - Écriture
  - Exécution
  - Concaténation (Append)
  - Suppression
  - List (voir les attributs)



Liste de contrôles d'accès

- La plupart des approches font dépendre les accès de l'identité de l'utilisateur
- Plus généralement, chaque fichier/répertoire contient une liste de contrôle d'accès :
  - contenue dans l'entrée du répertoire
  - Spécifie le nom de l'utilisateur et les droits



Protection sous UNIX

- Trois classifications par fichier :

- Possesseur (Owner) : l'utilisateur qui a créé le fichier
  - Peut être modifié avec ‘chown’ (par root)
- Groupe (Group) : un ensemble d'utilisateurs avec les même droits d'accès
  - Créé par root
  - Peut être modifié par root avec ‘chgrp’
- Autres (Universe) : tous les autres

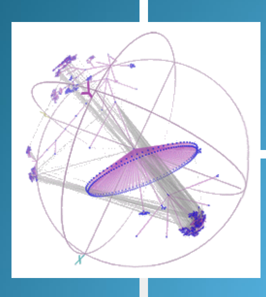


Protection sous UNIX (2)

- On peut spécifier les droits en lecture, écriture et exécution pour le possesseur, le groupe et les autres :

	RWX
possesseur	1 1 0 = 6
groupe	1 1 0 = 6
autres	0 0 0 = 0

- La commande ‘chmod’ permet de changer les droits
  - Ex : chmod 660 cours.ppt



Protection sous UNIX (3)

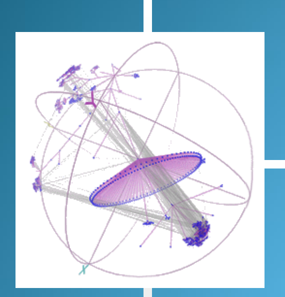
- Un exemple de répertoire :

```
$ ls -l
total 10547
drwxr-xr-x  1 ujgo7402 mkgroup-l-d       0 Feb  7 14:47 P2P_IPTPS
drwxr-xr-x  1 ujgo7402 mkgroup-l-d       0 Feb  6 19:27 PAPER
-rw-r--r--  1 ujgo7402 mkgroup-l-d 5117801 Feb  7 15:05 PAPER[1].tar.gz
-rw-r--r--  1 ujgo7402 mkgroup-l-d 5681444 Feb  7 15:04 iptps.tar.gz
```

- Répertoires :
  - R = lecture : lire le contenu du répertoire
  - W = écriture : créer ou supprimer des fichiers dans le répertoire
  - X = exécution : possibilité de se déplacer dans le répertoire

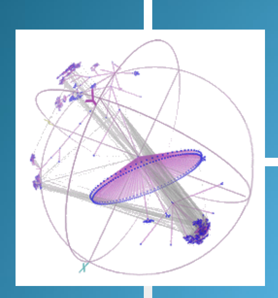


Implémentation des systèmes de fichiers



Introduction

- L'implémentation a pour but d'étudier :
  - Comment les fichiers sont stockés.
  - Comment les répertoires sont stockés.
  - Comment l'espace disque est géré.
  - Comment rendre le système de fichiers efficace.



Généralités

- Les systèmes de fichiers sont stockés sur disque.
- Les partitions contiennent des systèmes de fichiers indépendants.
- Plusieurs structures sont utilisées pour l'implémentation effective d'un système de fichiers. Ceci peut varier d'un système à l'autre.
- Le point crucial est de savoir en permanence quel fichier est associé à quel bloc sur le disque.

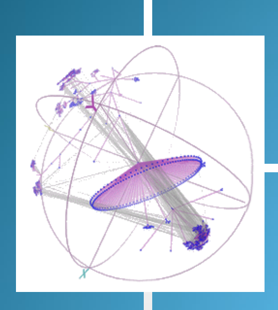


Blocs de contrôle

Sur le disque on trouve :

- Un bloc de contrôle de démarrage (boot control block)
  - contient les informations nécessaire pour démarrer le SE depuis cette partition.
  - Typiquement le premier bloc de la partition.
  - Sous UNIX (UFS) : boot block.
  - Avec NTFS : partition boot block.

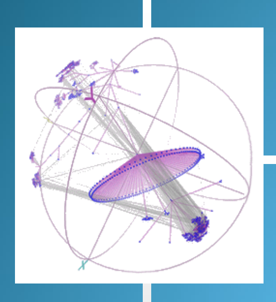
- Un bloc de contrôle de la partition
  - Contient des informations sur la partition (nombre de blocs, taille des blocs, nombre de blocs libres, pointeurs sur les blocs libres).
  - UFS : superblock.
  - NTFS : Master file table.



Bloc de contrôle d'un fichier

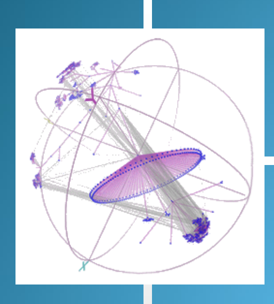
- Le bloc de contrôle d'un fichier (FCB) contient les informations sur le fichier (permissions, taille, etc.)

- UFS : inode.
- NTFS : stocké dans la Master File Table.



Méthodes d'accès

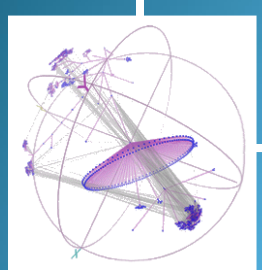
Séquentielle
Indexée
Directe

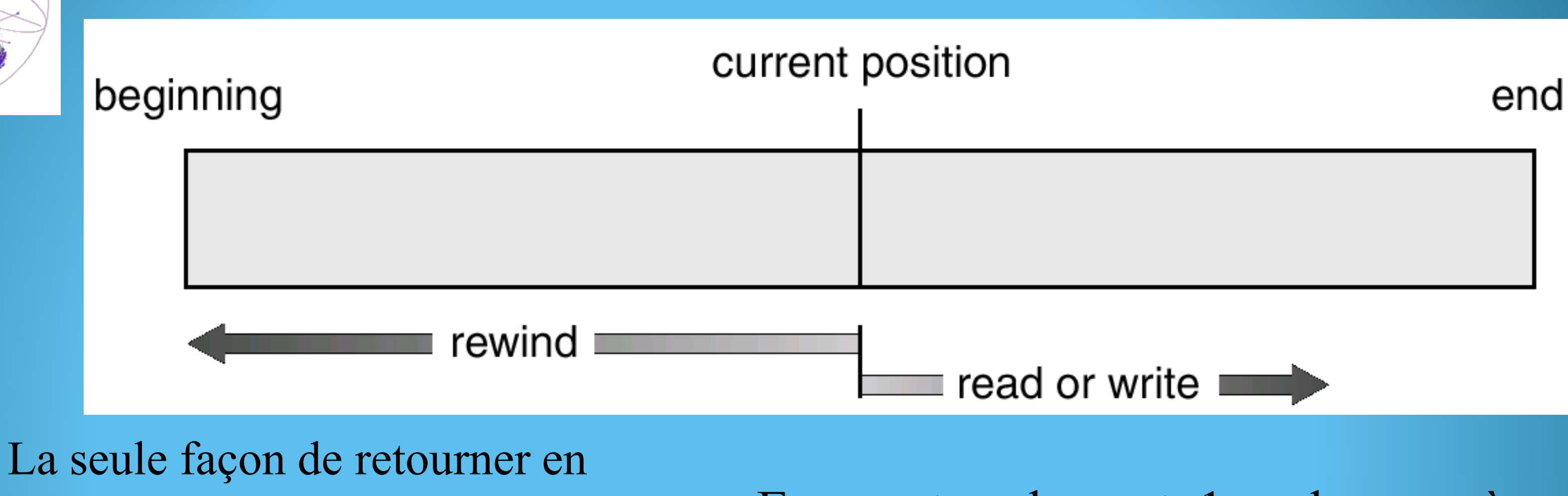


Méthodes d’accès: 4 de base

- Séquentiel (rubans ou disques): lecture ou écriture des enregistrements dans un ordre fixe
- Indexé séquentiel (disques): accès séquentiel ou accès direct (aléatoire) par l’utilisation d’index
- Indexée: multiplicité d’index selon les besoins, accès direct par l’index
- Direct ou hachée: accès direct à travers tableau d’hachage

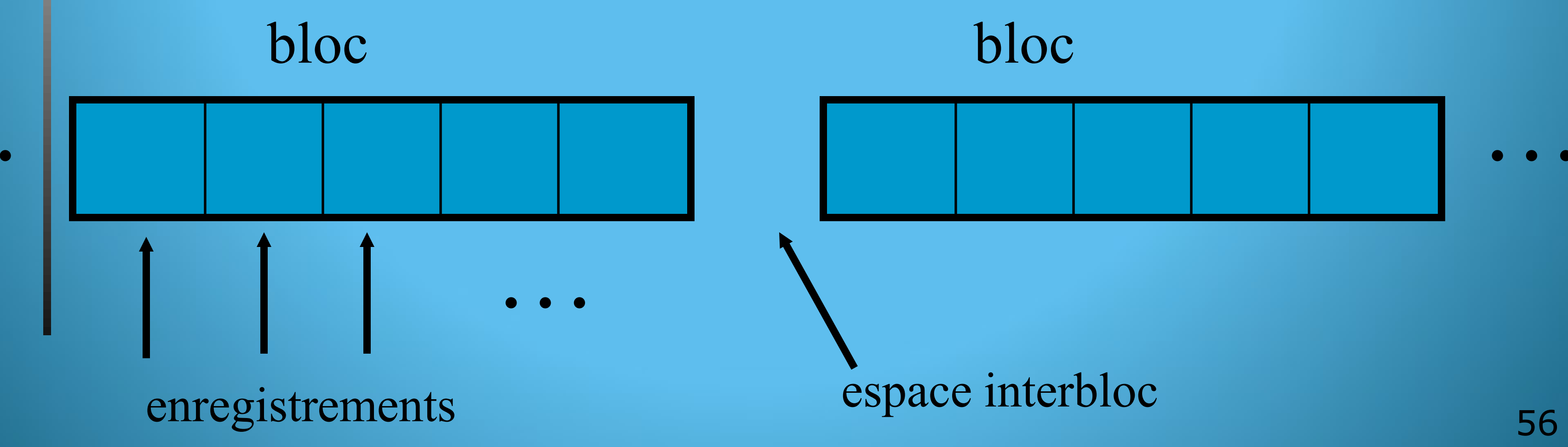
Fichiers à accès séquentiel (rubans)

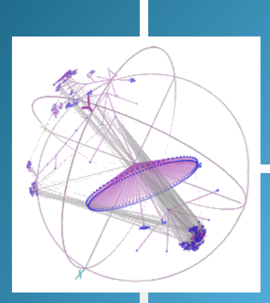




La seule façon de retourner en arrière est de retourner au début (rébobiner, rewind)

En avant seulement, 1 seul enreg. à la fois





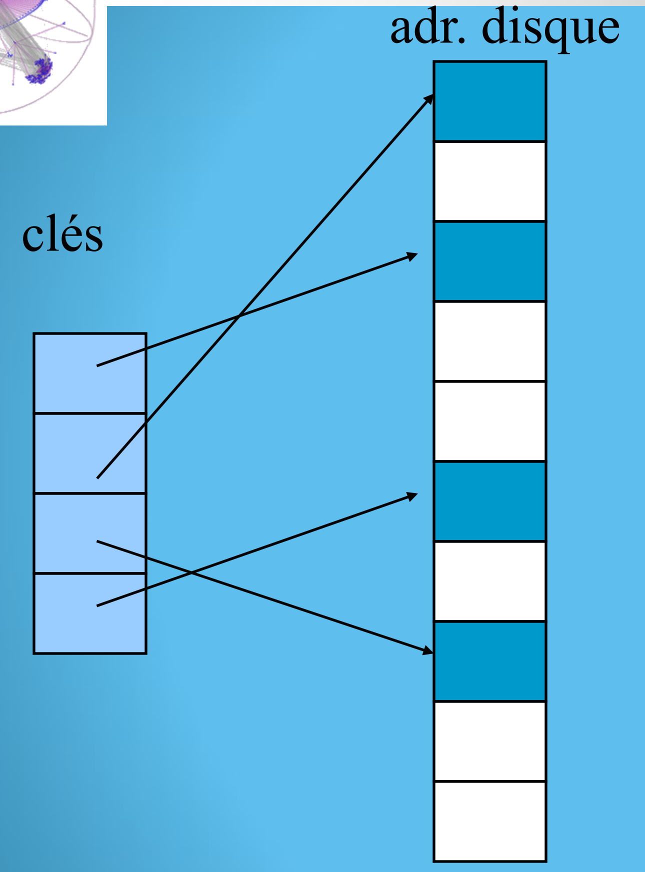
Accès direct ou haché ou aléatoire: accès direct à travers tableau d'hachage

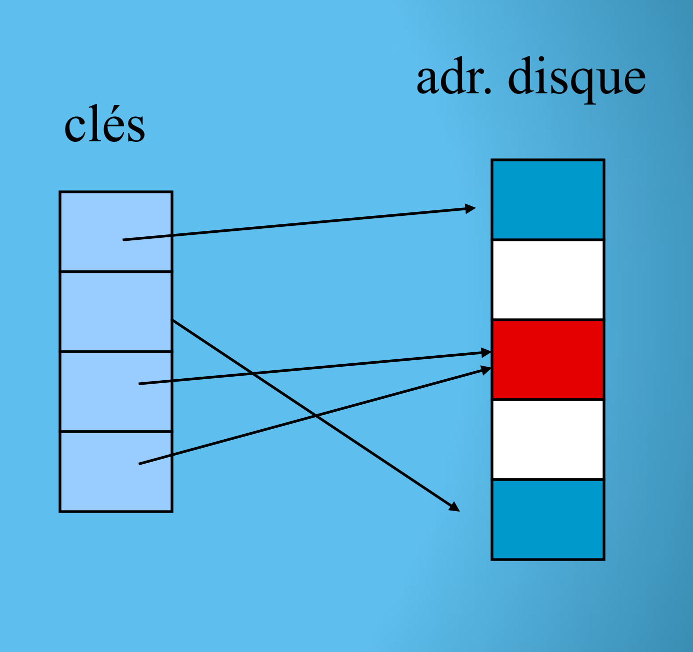
- Une fonction d'hachage est une fonction qui traduit une clé dans adresse,
  - P.ex. Matricule étudiant → adresse disque

- Rapide mais:
  - Si les adresses générées sont trop éparpillées, gaspillage d'espace
  - Si les adresses ne sont pas assez éparpillées, risque que deux clés soient renvoyées à la même adresse
    - Dans ce cas, il faut de quelques façon renvoyer une des clés à une autre adresse



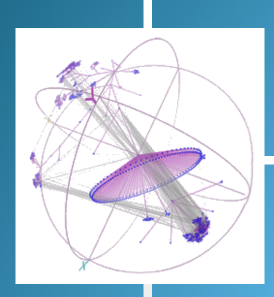
Problème avec les fonctions d'hachage





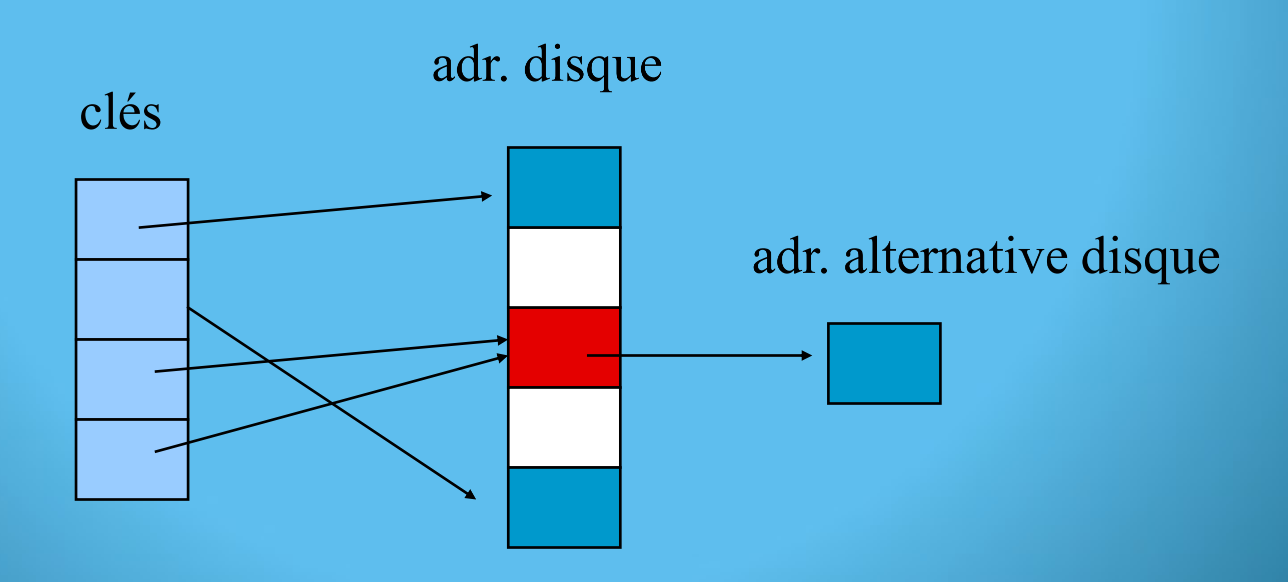
Fonction d'hachage dispersée qui n'utilise pas bien l'espace disponible

Fonction d'hachage concentrée qui utilise mieux l'espace mais introduit des doubles affectations



Hachage: Traitement des doubles affectations

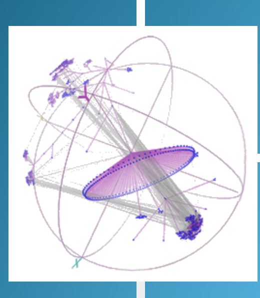
- On doit détecter les doubles affectations, et s'il y en a, un des deux enregistrements doit être mis ailleurs
  - ce qui complique l'algorithme





Adressage Indexé séquentiel

- **Un index permet d'arriver directement à l'enregistrement désiré, ou en sa proximité**
  - Chaque enregistrement contient un champ clé
  - Un fichier index contient des repères (pointeurs) à certain points importants dans le fichier principal (p.ex. début de la lettre S, début des Lalande, début des matricules qui commencent par 8)
  - Le fichier index pourra être organisé en niveaux (p.ex. dans l'index de S on trouve l'index de tous ceux qui commencent par S)
  - Le fichier index permet d'arriver au point de repère dans le fichier principal, puis la recherche est séquentielle



Exemples d'index et fichiers relatifs

logical record number
last name

Adams
Arthur
Asher
Smith

Smith, John | social-security | age

index file
relative file

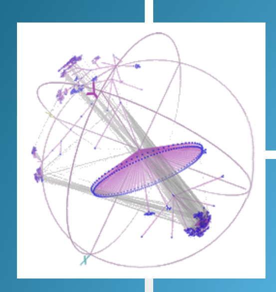
Pointe au début des Smiths (il y en aura plusieurs)

Le fichier index est à accès direct, le fichier relatif est à accès séquentiel

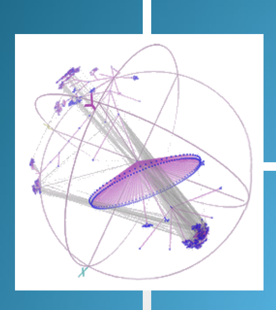


Comparaison : Séquentiel et index séquentiel

- **P.ex. Un fichier contient 1 million d'enregistrements**
- **En moyenne, 500.000 accès sont nécessaires pour trouver un enregistrement si l'accès est séquentiel!**
- **Mais dans un séquentiel indexé, s'il y a un seul niveau d'indices avec 1000 entrées (et chaque entrée pointe donc à 1000 autres),**
- ***En moyenne*, ça prend 1 accès pour trouver le repère approprié dans le fichier index**
- **Puis 500 accès pour trouver séquentiellement le bon enregistrement dans le fichier relatif**



Méthodes d’allocation

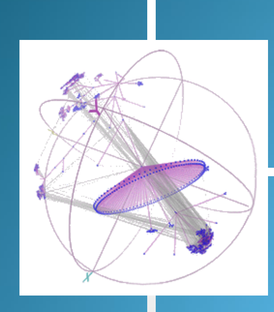


Structures de systèmes de fichiers

- Le système de fichiers réside dans la mémoire secondaire: disques, rubans...
- File control block: structure de données contenant de l’info sur un fichier (RAM)

Structure physique des fichiers

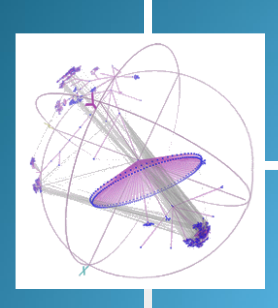
- **La mémoire secondaire est subdivisée en blocs et chaque opération d'E/S s'effectue en unités de blocs**
  - Les blocs ruban sont de longueur variable, mais les blocs disque sont de longueur fixe
  - Sur disque, un bloc est constitué d'un multiple de secteurs contiguës (ex: 1, 2, ou 4)
    - la taille d'un secteur est habituellement 512 octets
- **Il faut donc insérer les enregistrements dans les blocs et les extraire par la suite**
  - Simple lorsque chaque octet est un enregistrement par lui-même
  - Plus complexe lorsque les enregistrements possèdent une structure
- **Les fichiers sont alloués en unité de blocs. Le dernier bloc est donc rarement rempli de données**
  - Fragmentation interne



Méthodes d'allocation

Comment les blocs sur le disque sont alloués pour les fichiers. Il existe plusieurs méthodes, dont :

- Allocation contiguë
- Allocation liée
- Allocation indexée



Allocation contiguë

- Chaque fichier occupe des blocs contigus sur le disque.

Avantages

- Simplicité : il suffit de connaître la position (numéro du bloc) et la longueur du fichier (nombre de blocs).
- Accès aléatoire à l'information.

Inconvénients

- Perte d'espace disque (problèmes de fragmentation).
- Les fichiers ne peuvent pas grandir.



Allocation contiguë (2)

count

0 1 2 3

f

4 5 6 7

8 9 10 11

tr

12 13 14 15

16 17 18 19

mail

20 21 22 23

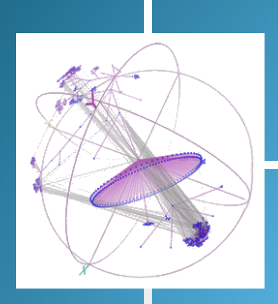
24 25 26 27

list

28 29 30 31

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2



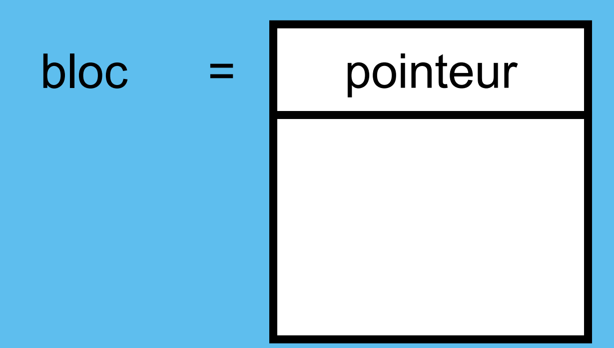
Allocation contiguë

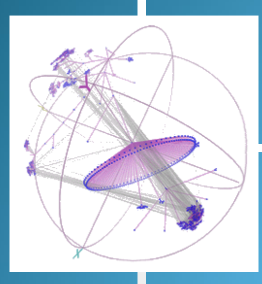
- Chaque fichier occupe un ensemble de blocs contiguë sur disque
- Simple: nous n’avons besoin que d’adresses de début et longueur
- Supporte tant l’accès séquentiel, que l’accès direct
- Application des problèmes et méthodes vus dans le chapitre de l’allocation de mémoire contiguë
- Les fichiers ne peuvent pas grandir
- Impossible d’ajouter au milieu
- Exécution périodique d’une compression (compaction) pour récupérer l’espace libre



Allocation enchaînée

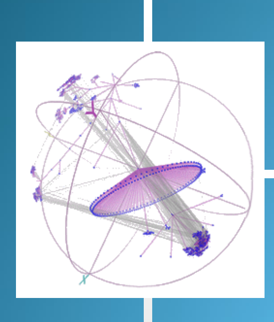
- Le répertoire contient l’adresse du premier et dernier bloc, possible le nombre de blocs
- Chaque bloc contient un pointeur à l’adresse du prochain bloc:





Allocation enchaînée

file	start	end
jeep	9	25



Avantages - désavantages

- Pas de fragmentation externe - allocation de mémoire simple, pas besoin de compression
- L'accès à l'intérieur d'un fichier ne peut être que séquentiel
  - Pas de façon de trouver directement le 4ème enregistrement...
- L'intégrité des pointeurs est essentielle
- Les pointeurs gaspillent un peu d'espace



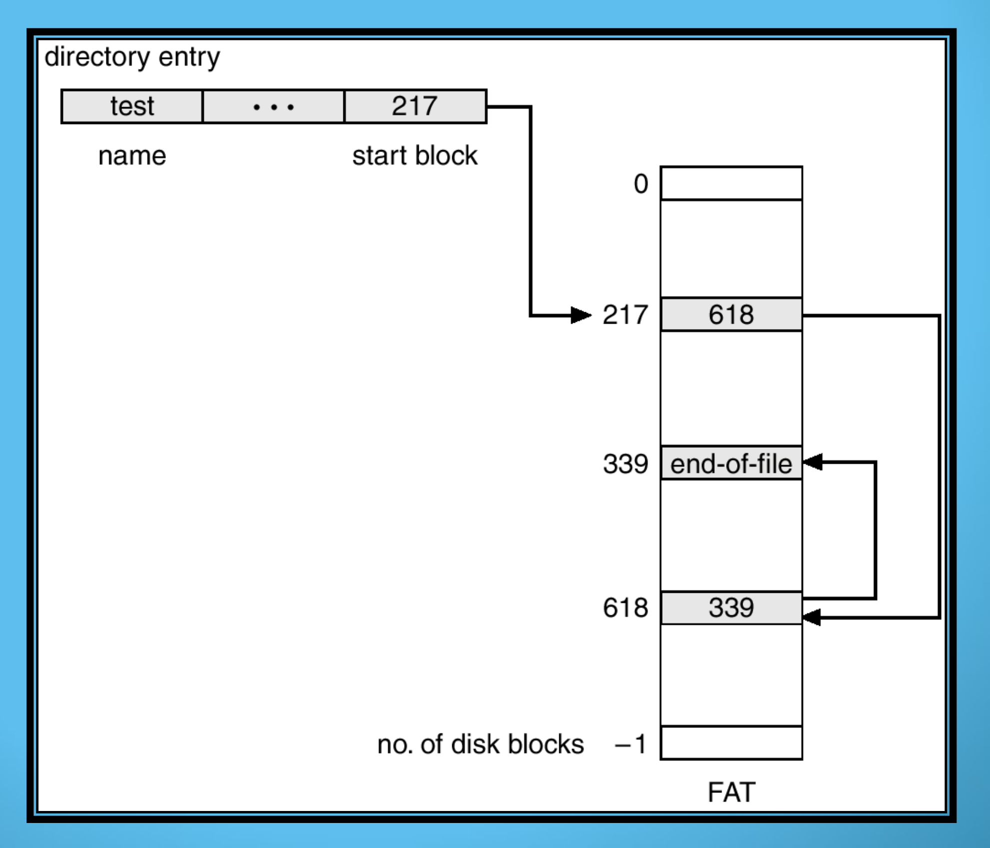
Table d'allocation des fichiers

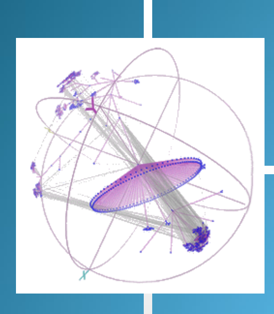
- Une variation de l'allocation liée est d'utiliser une table d'allocation des fichiers (FAT).
  - Utilisé par MS-DOS et OS/2.
  - Une partie du disque est réservée pour stocker la table qui contient les pointeurs vers tous les fichiers de la partition.
  - Chaque entrée dans la FAT correspond à un bloc sur le disque. Chaque entrée contient le pointeur vers le bloc suivant du fichier.
  - Une valeur spéciale indique la fin du fichier.
  - Une entrée nulle signifie un bloc inutilisé.



Table d'allocation

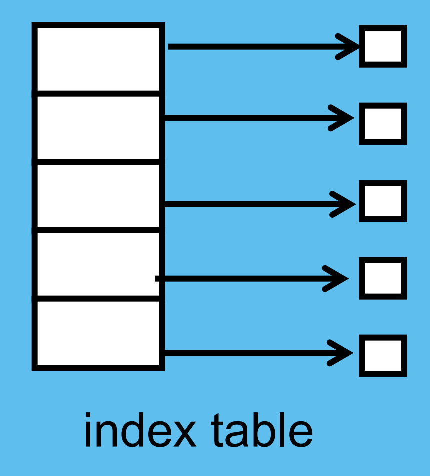
- Les FATs sont stockées en mémoire tant que le SE est actif

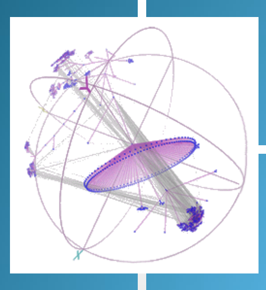




Allocation indexée: semblable à la pagination

- Tous les pointeurs sont regroupés dans un tableau (index block)





Allocation indexée

directory

file	index block
jeep	19

0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31

19

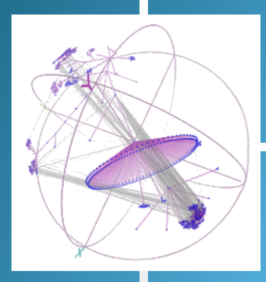
9
16
1
10
25
–1
–1
–1

-1: pointeur nul

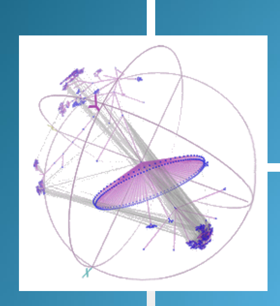


Allocation indexée

- À la création d’un fichier, tous les pointeurs dans le tableau sont *nil (-1)*
- Chaque fois qu’un nouveau bloc doit être alloué, on trouve de l’espace disponible et on ajoute un pointeur avec son adresse
- Pas de fragmentation externe, mais les index prennent de l’espace
- Permet accès direct (aléatoire)
- Taille de fichiers limitée par la taille de l’index block
  - Mais nous pouvons avoir plusieurs niveaux d’index: Unix
- Index block peut utiliser beaucoup de mémoire



UNIX BSD: indexé à niveaux (config. possible)



Concepts communs

- Les fichiers sont représentés par des inodes
- Les répertoires sont des fichiers
- Les périphériques sont accédés par des entrées sorties sur des fichiers spéciaux (/dev/hda0)
- Les liens sont autorisés

Inodes

- Une structure qui contient la description du fichier :
  - Type
  - Droits d'accès
  - Possesseurs
  - Dates de modifications
  - Taille
  - Pointeurs vers les blocs de données
- Les inodes sont stockés en mémoire tant que le fichier est ouvert



Inodes (2)

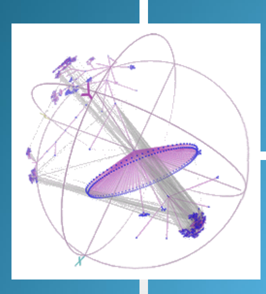
inode

infos

Blocs directs

Indirection simple

Indirection double



i-noeud

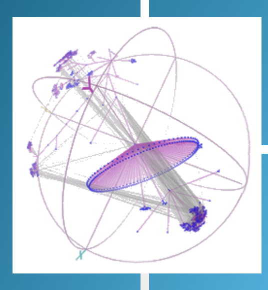
Attributs

10 blocs

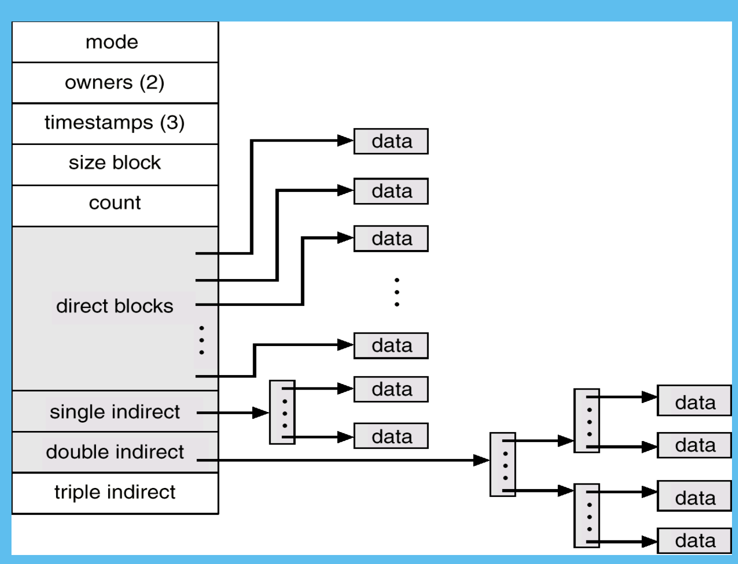
Simple ind.

Double ind.

Triple ind.

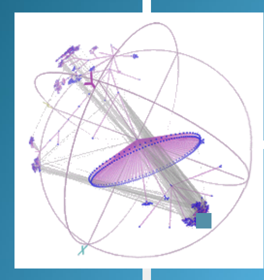


UNIX BSD

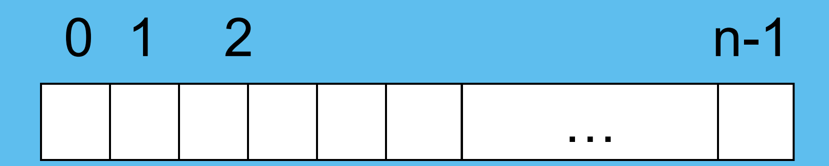


- Les premiers blocs d'un fichier sont accessibles directement
- Si le fichier contient des blocs additionnels, les premiers sont accessibles à travers un niveau d'indices
- Les suivants sont accessibles à travers 2 niveaux d'indices, etc.
- Donc le plus loin du début un enregistrement se trouve, le plus indirect est son accès
- Permet accès rapide à petits fichiers, et au début de tous les fichiers.
- Permet l'accès à des grands fichier avec un petit répertoire en mémoire

Gestion d'espace libre



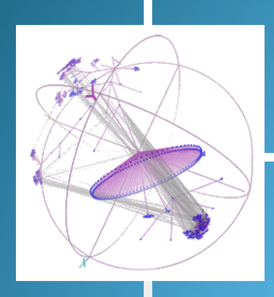
- Vecteur de bits (*n* blocs)



bit[*i*] = { 0 ⇒ block[*i*] libre ; 1 ⇒ block[*i*] occupé }

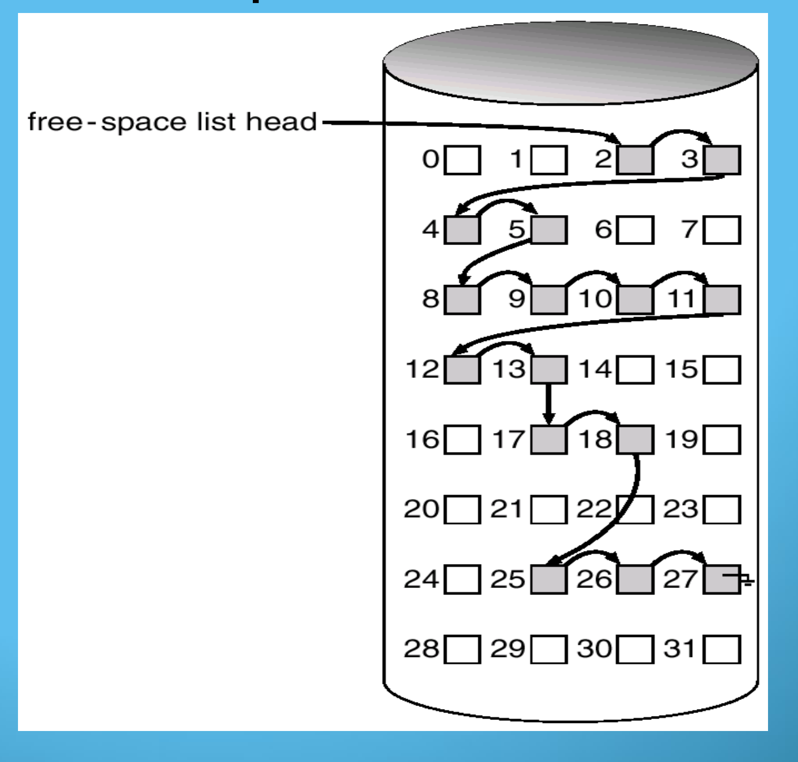
- **Exemple d'un vecteur de bits où les blocs 3, 4, 5, 9, 10, 15, 16 sont occupés:**
  - 00011100011000011…
- **L'adresse du premier bloc libre peut être trouvée par un simple calcul**

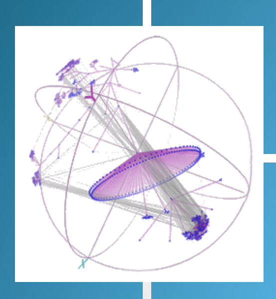
Gestion d’espace libre



Solution 2: Liste liée de mémoire libre (MS-DOS, Windows 9x)

Tous les blocs de mémoire libre sont liés ensemble par des pointeurs





Comparaison

- Bitmap:
  - si la bitmap de toute la mémoire secondaire est gardée en mémoire principale, la méthode est rapide mais demande de l'espace de mémoire principale
  - si les bitmaps sont gardées en mémoire secondaire, temps de lecture de mémoire secondaire...
    - Elles pourraient être paginées, p.ex.
- Liste liée
  - Pour trouver plusieurs blocs de mémoire libre, plus d'accès disques pourraient être demandés
  - Pour augmenter l'efficacité, nous pouvons garder en mémoire centrale l'adresse du 1er bloc libre



Structure de mémoire de masse (disques)

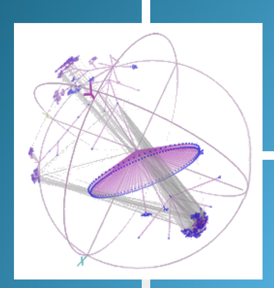
Concepts importants :

- **Fonctionnement et structure des unités disque**
- **Calcul du temps d'exécution d'une séquence d'opérations**
- **Différents algorithmes d'ordonnancement**
  - Fonctionnement, rendement
- **Gestion de l'espace de permutation**
  - Unix



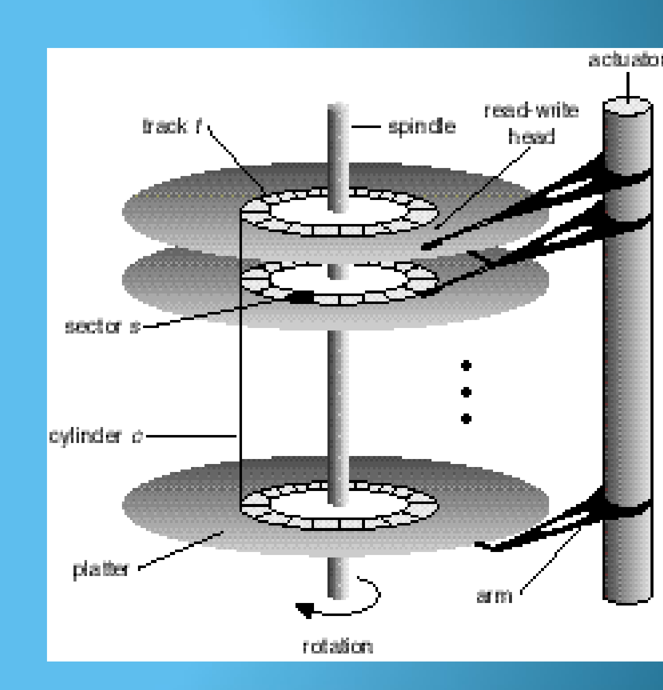
Ordonnancement disques

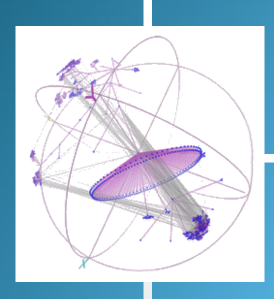
- Problème: utilisation optimale du matériel
- Réduction du temps total de lecture disque
  - étant donné une file de requêtes de lecture disque, dans quel ordre les exécuter?



Paramètres à prendre en considération

- Temps de positionnement (seek time):
  - le temps pris par l`unité disque pour se positionner sur le cylindre désiré
- Temps de latence de rotation (latency time)
  - le temps pris par l 'unité de disque qui est sur le bon cylindre pour se positionner sur le secteur désirée
- Temps de lecture
  - temps nécessaire pour lire la piste
- Le temps de positionnement est normalement le plus important, donc il est celui que nous chercherons à minimiser





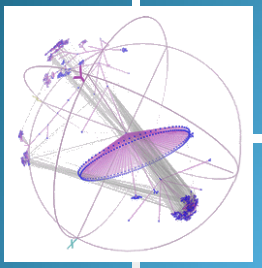
File d’attente disque

- Dans un système multiprogrammé avec mémoire virtuelle, il y aura normalement une file d’attente pour l’unité disque
- Dans quel ordre choisir les requêtes d’opérations disques de façon à minimiser les temps de recherche totaux
- Nous étudierons différents méthodes par rapport à une file d’attente arbitraire:

98, 183, 37, 122, 14, 124, 65, 67

- Chaque chiffre est un numéro séquentiel de cylindre
- Il faut aussi prendre en considération le cylindre de départ: 53
- Dans quel ordre exécuter les requêtes de lecture de façon à minimiser les temps totaux de positionnement cylindre
- Hypothèse simpliste: un déplacement d`1 cylindre coûte 1 unité de temps

Premier entré, premier sorti: FIFO



queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

0 14 37 53 65 67 98 122 124 183 199

45
85
146
85
108
110
59
2

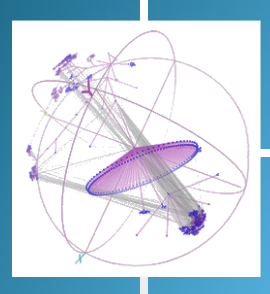
Mouvement total: 640 cylindres = (98-53) + (183-98)+...
En moyenne: 640/8 = 80



SSTF: Shortest Seek Time First

Plus Court Temps de Recherche (positionnement) d'abord (PCTR)

- À chaque moment, choisir la requête avec le temps de recherche le plus court à partir du cylindre courant
- Clairement meilleur que le précédent
- Mais pas nécessairement optimal!
- Peut causer famine



SSTF: Plus court servi (PCTR)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

0 14 37 53 65 67 98 122 124 183 199

Mouvement total: 236 cylindres (680 pour le précédent)

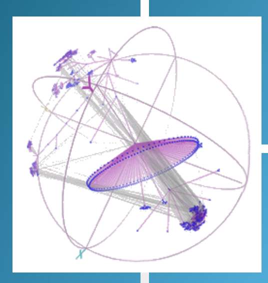
En moyenne: 236/8 = 29.5 (80 pour le précédent)



LOOK: l'algorithme de l'ascenseur
SCAN: l'algorithme du bus

- Scan : La tête balaye le disque dans une direction, puis dans la direction opposée, jusqu'au bout. etc., en desservant les requêtes quand il passe sur le cylindre désiré
  - Pas de famine
- Look : La tête balaye le disque dans une direction jusqu'il n'y aie plus de requête dans cette direction, puis dans la direction opposée de même, etc., en desservant les requêtes quandil passe sur le cylindre désiré

SCAN: l’ascenseur

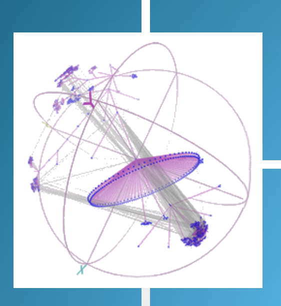


direction ←

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

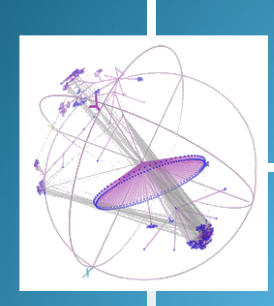
0 14 37 53 65 67 98 122 124 183 199

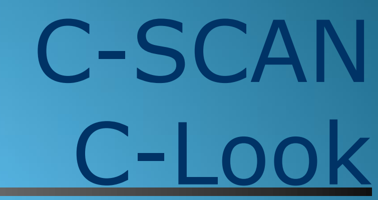
Mouvement total: 208 cylindres
En moyenne: 208/8 = 26 (29.5 pour SSTF)



Problèmes du SCAN

- Peu de travail à faire après le renversement de direction
- Les requêtes seront plus denses à l'autre extrémité
- Arrive inutilement jusqu'à 0





- Retour rapide au début (cylindre 0) du disque au lieu de renverser la direction
- Hypothèse: le mécanisme de retour est beaucoup plus rapide que le temps de visiter les cylindres
  - Comme si les disques étaient en forme de beignes!

C-LOOK

- La même idée, mais au lieu de retourner au cylindre 0, retourner au premier cylindre qui a une requête

C-LOOK



direction →

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

0 14 37 53 65 67 98 122 124 183 199

retour: 169 (??)

153 sans considérer le retour (19.1 en moyenne) (26 pour SCAN)
MAIS 322 avec retour (40.25 en moyenne)
Normalement le retour sera rapide donc le coût réel sera entre les deux

C-LOOK avec direction initiale opposée



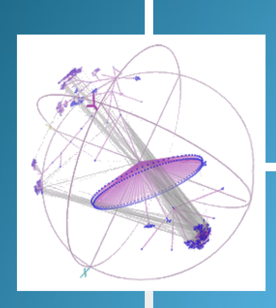
queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53 direction ←

0 14 37 53 65 67 98 122 124 183 199

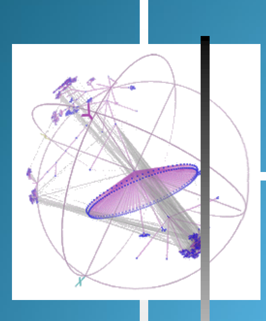
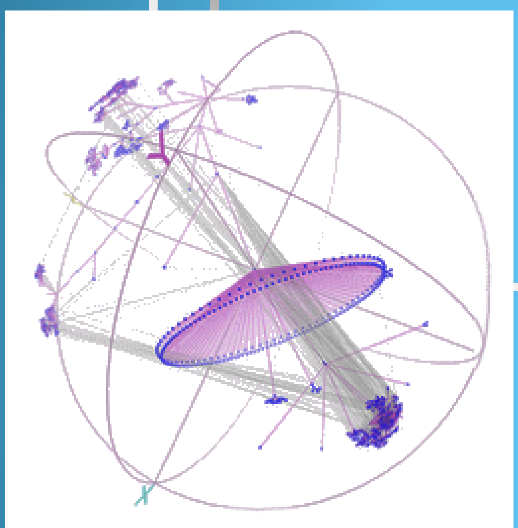
Retour 169

Résultats très semblables:
157 sans considérer le retour, 326 avec le retour



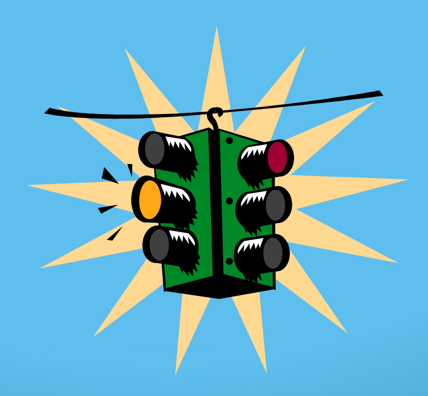
Comparaison

- Si la file souvent ne contient que très peu d'éléments, l'algorithme du 'premier servi ' devrait être préféré (simplicité)
- Sinon, SSTF ou SCAN ou C-SCAN?
- En pratique, il faut prendre en considération:
  - Les temps réels de déplacement et retour au début
  - L`organisation des fichiers et des répertoires
    - Les répertoires sont sur disque aussi…
  - La longueur moyenne de la file
  - Le débit d 'arrivée des requêtes

Synchronisation de Processus

Chapitre 5



Synchronisation de Processus

1. **Conditions de Concurrence**
2. **Sections Critiques**
3. **Exclusion Mutuelle**
4. **Sommeil & Activation**
5. **Sémaphores**
6. **Mutex**

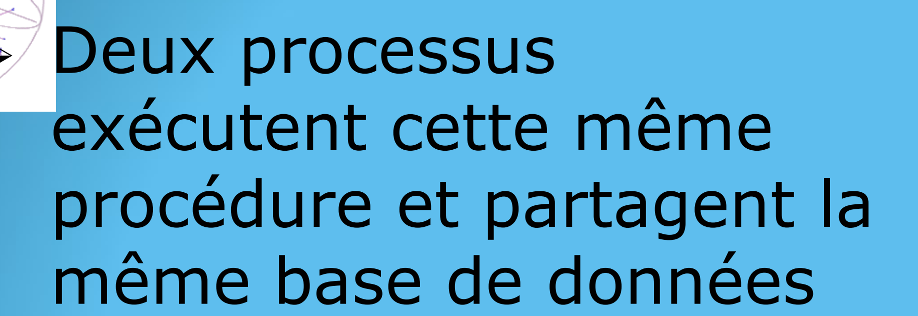


Concurrence

- Les processus concurrents doivent parfois partager données (fichiers ou mémoire commune) et ressources
  - On parle donc de tâches *coopératives*
- Si l’accès n’est pas contrôlé, le résultat de l’exécution du programme pourra dépendre de l’ordre d’entrelacement de l’exécution des instructions (*non-déterminisme*).
- Un programme pourra donner des résultats différents et parfois indésirables



Un exemple

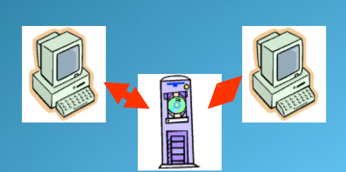


- Deux processus exécutent cette même procédure et partagent la même base de données
- Ils peuvent être interrompus n'importe où
- Le résultat de l'exécution concurrente de P1 et P2 dépend de l'ordre de leur *entrelacement*

M. X demande une réservation d'avion

Base de données dit que fauteuil A est disponible

Fauteuil A est assigné à X et marqué occupé





Exemple d'exécution possible

P1

P2

Interruption ou retard

M. Leblanc demande une réservation d'avion

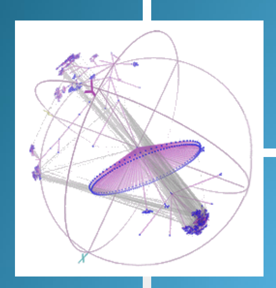
M. Guy demande une réservation d'avion

Base de données dit que fauteuil 30A est disponible

Base de données dit que fauteuil 30A est disponible

Fauteuil 30A est assigné à Leblanc et marqué occupé

Fauteuil 30A est assigné à Guy et marqué occupé



Un autre exemple

P1	interruption	P2
`b=a`	→	
		`b=a`
		`b++`
	←	`a=b`
`b++`		
`a=b`		

Deux opérations en parallèle sur une variable a partagée (b est privé à chaque processus)

Supposons que a soit 0 au début

P1 travaille sur le vieux a donc le résultat final sera a=1.

Il serait a=2 si les deux tâches sont exécutées l'une après l'autre

Si a était sauvegardé quand P1 est interrompu, il ne pourrait pas être partagé avec P2 (il y aurait deux a tandis que nous en voulons une seule)



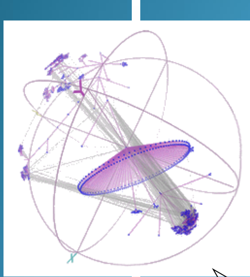
Section Critique

- Partie d'un programme dont l'exécution ne doit pas *entrelacer* avec autres programmes
- Une fois qu'un tâche y entre, il faut lui permettre de terminer cette section sans permettre à autres tâches de jouer sur les mêmes données

Le problème de la section critique

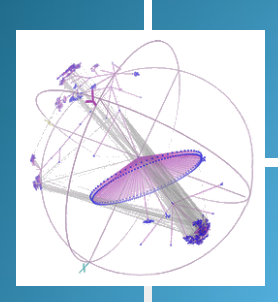
- Lorsqu’un processus manipule une donnée (ou ressource) partagée, nous disons qu’il se trouve dans une section critique (SC) (associée à cette donnée)
- Le problème de la section critique est de trouver un algorithme d`exclusion mutuelle de processus dans l’exécution de leur Section Critiques afin que le résultat de leurs actions ne dépendent pas de l’ordre d’entrelacement de leur exécution (avec un ou plusieurs processeurs)
- L’exécution des sections critiques doit être mutuellement exclusive: à tout instant, un seul processus peut exécuter une SC pour une var donnée (même lorsqu’il y a plusieurs processeurs)
- Ceci peut être obtenu en plaçant des instructions spéciales dans les sections d`entrée et sortie
- Pour simplifier, dorénavant nous faisons l’hypothèse qu’il n’y a qu'une seule SC dans un programme.

Structure du programme



- Chaque processus doit donc demander une permission avant d'entrer dans une section critique (SC)
- La section de code qui effectue cette requête est la section d'entrée
- La section critique est normalement suivie d'une section de sortie
- Le code qui reste est la section restante (SR): non-critique

```
repeat
   section d'entrée
    section critique
   section de sortie
    section restante
forever
```



Application

M. X demande une réservation d'avion

Section critique

Section d'entrée

Base de données dit que fauteuil A est disponible

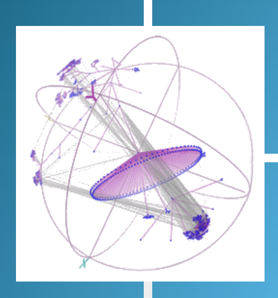
Fauteuil A est assigné à X et marqué occupé

Section de sortie



Critères nécessaires pour solutions valides

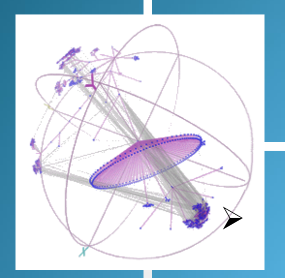
- Exclusion Mutuelle
  - À tout instant, au plus un processus peut être dans une section critique (SC) pour une variable donnée
- Non interférence:
  - Si un processus s’arrête dans sa section restante, ceci ne devrait pas affecter les autres processus
- Mais on fait l’hypothèse qu’un processus qui entre dans une section critique, en sortira.



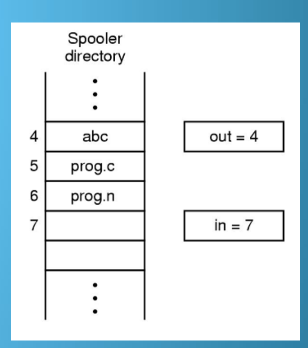
Critères nécessaires pour solutions valides

- Progrès:
  - absence d’interblocage
  - si un processus demande d`entrer dans une section critique à un moment où aucun autre processus en fait requête, il devrait être en mesure d’y entrer
- Absence de famine: aucun processus ne sera éternellement empêché d’atteindre sa Section Critique

Conditions de Concurrence



- Conditions de concurrence (*race conditions*): situation où 2 processus ou plus effectuent des lectures et des écritures conflictuelles.
- Exemple du Spouler d'impression
  - Un processus qui veut imprimer un fichier, entre son nom dans un **répertoire de spoule**
  - Le processus **démon d'impression** regarde périodiquement s'il y a des fichiers à imprimer. Il a 2 variables:
    - **in**: pointe vers la prochaine entrée libre.
    - **out**: pointe vers le prochain fichier à imprimer
  - *in* = 7, *out* = 4
  - A et B deux processus qui veulent imprimer un fichier
  - A >> lire *in*, *next_free_slot* = 7
  - Interruption: la CPU bascule vers le processus B
  - B >> lire *in*, *next_free_slot* = 7, entrée7 = fichierB, *in* = 8
  - A >> entrée7 = fichierA, *in* = 8
  - Problème: le fichierB ne sera pas imprimé



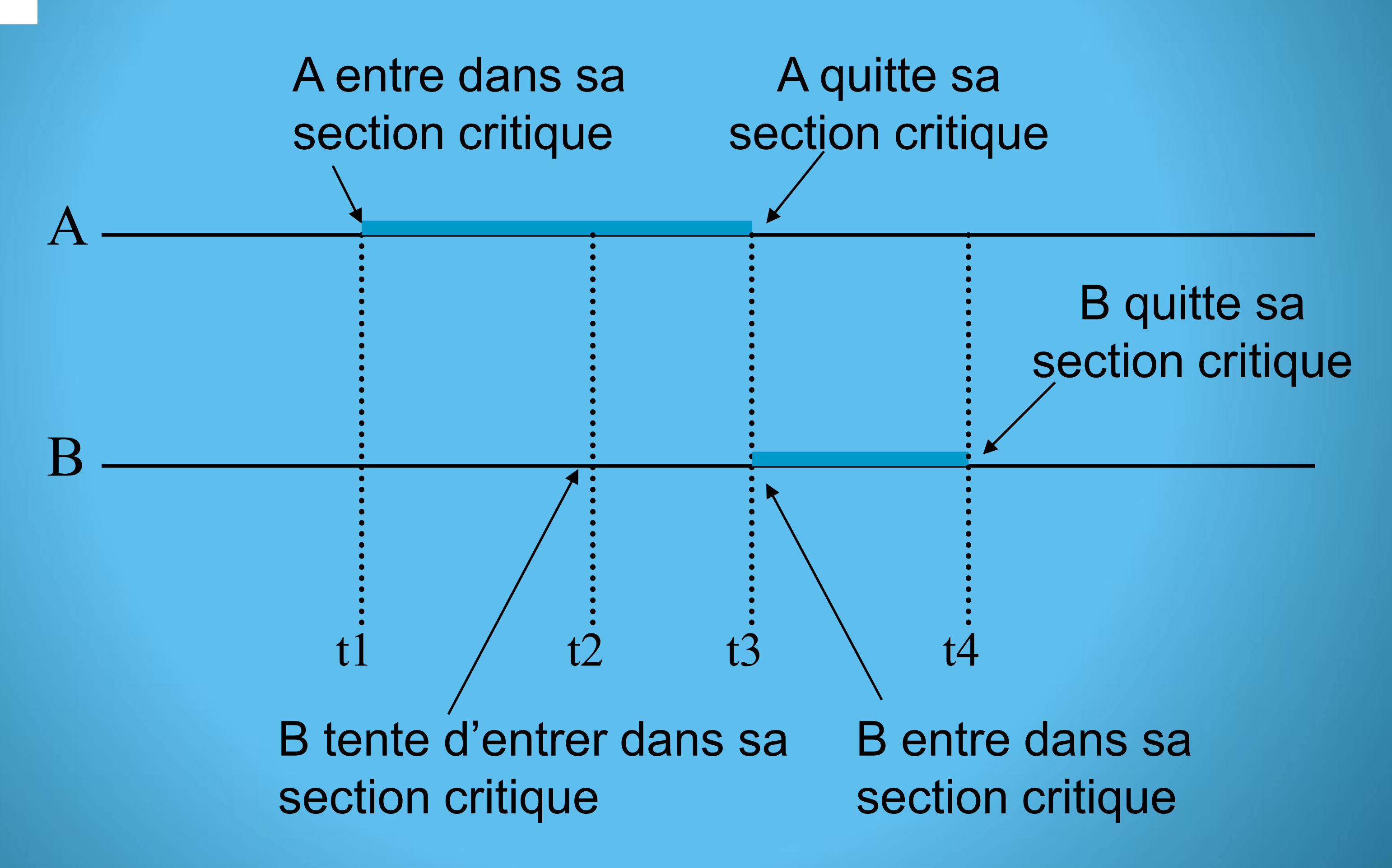


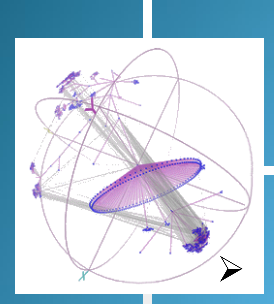
… Conditions de Concurrence

- Comment éviter les conditions de concurrence?
- Solution: Interdire que plusieurs processus lisent et écrivent des données partagées simultanément.
- Exclusion Mutuelle: permet d'assurer que si un processus utilise une variable ou fichier partagés, les autres processus seront exclus de la même activité

Les Sections Critiques

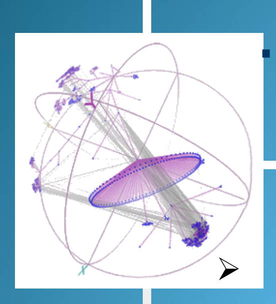
- les Sections Critiques, méthode d'exclusion mutuelle





L'Exclusion Mutuelle avec Attente Active (*busy waiting*)

- Désactivation des interruptions
  - Après son entrée dans une SC, un processus désactive les interruptions, puis les réactive
  - Il empêche ainsi l'horloge d'envoyer des interruptions et le processeur de basculer
  - Il est imprudent de permettre à des processus user de désactiver les interruptions
- Variables de verrou *(lock)*
  - Avant d'entrer en SC, tester la valeur de verrou, si verrou = 0, verrou ← 1, entrer en SC
  - Défaillance: 2 processus peuvent entrer simultanément dans leurs sections critiques comme le spouler d'impression
- Alternance Stricte
  - la variable *turn* porte le numéro du processus dont c'est le tour d'entrer en SC. Chaque processus inspecte la valeur de la variable, avant d'entrer en SC.
  - Inconvénient: consomme bcp de temps CPU



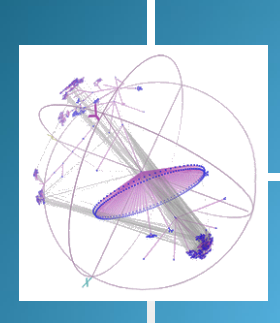
… Exclusion Mutuelle avec Attente Active (*busy waiting*)

… Alternance Stricte

```
while (TRUE) {
    while (turn != 0);
    critical_region();
    turn = 1;
    non_critical_region();
}
```

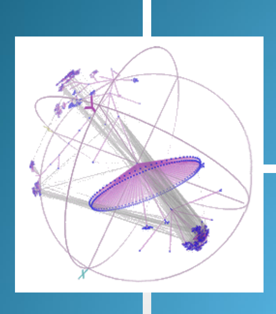
```
while (TRUE) {
    while (turn != 1);
    critical_region();
    turn = 0;
    non_critical_region();
}
```

- Les attentes actives sont performantes dans le cas où elles sont brèves. En effet, il y' a risque d'attente
  - P0 quitte la CS, turn = 1
  - P1 termine sa CS, turn = 0
  - Les 2 processus sont en section non critique
  - P0 exécute sa boucle, quitte la SC et turn = 1
  - Les 2 processus sont en section non critique
  - P0 quoiqu'il a terminé, il ne peut pas entrer en SC, il est bloqué



Une leçon à retenir…

- À fin que des processus avec des variables partagées puissent réussir, il est nécessaire que tous les processus impliqués utilisent le même algorithme de coordination
  - Un protocole commun



Critique des solutions par logiciel

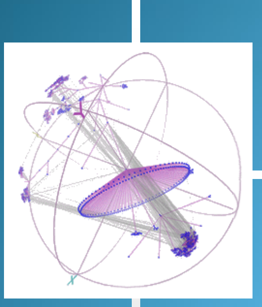
- Difficiles à programmer! Et à comprendre!
  - Les solutions que nous verrons dorénavant sont toutes basées sur l'existence d'instructions spécialisées, qui facilitent le travail.

- Les processus qui requièrent l'entrée dans leur SC sont occupés à attendre (busy waiting); consommant ainsi du temps de processeur
  - Pour de longues sections critiques, il serait préférable de bloquer les processus qui doivent attendre...

Solutions matérielles: désactivation des interruptions

- Sur un uniprocesseur: exclusion mutuelle est préservée mais l’efficacité se détériore: lorsque dans SC il est impossible d’entrelacer l’exécution avec d’autres processus dans une SR
- Perte d’interruptions
- Sur un multiprocesseur: exclusion mutuelle n’est pas préservée
- Une solution qui n’est généralement pas acceptable

```
Process Pi:
repeat
   inhiber interrupt
    section critique
   rétablir interrupt
    section restante
forever
```



Solutions basées sur des instructions fournies par le SE (appels du système)

- Les solutions vues jusqu'à présent sont difficiles à programmer et conduisent à du mauvais code.
- On voudrait aussi qu`il soit plus facile d'éviter des erreurs communes, comme interblocages, famine, etc.
  - Besoin d'instruction à plus haut niveau
- Les méthodes que nous verrons dorénavant utilisent des instructions puissantes, qui sont implantées par des appels au SE (system calls)

Sémaphores

- Un sémaphore S est un entier qui, sauf pour l'Initialisation, est accessible seulement par ces 2 opérations atomiques et mutuellement exclusives:
  - wait(S)
  - signal(S)
- Il est partagé entre tous les procs qui s`intéressent à la même section critique
- Les sémaphores seront présentés en deux étapes:
  - sémaphores qui sont occupés à attendre (busy waiting)
  - sémaphores qui utilisent des files d ’attente
- On fait distinction aussi entre sémaphores compteurs et sémaphores binaires, mais ce derniers sont moins puissants.

Sémaphores occupés à attendre (busy waiting)

- La façon la plus simple d'implanter les sémaphores.
- Utiles pour des situations où l'attente est brève, ou il y a beaucoup d'UCTs
- S est un entier initialisé à une valeur positive, de façon que un premier processus puisse entrer dans la SC
- Quand S>0, jusqu'à n processus peuvent entrer
- Quand S<=0, il faut attendre S+1 signals (d'autres processus) pour entrer

```
wait(S):
while S<=0 {};
 S--;
```

Attend si no. de processus qui peuvent entrer = 0 ou négatif

```
signal(S):
 S++;
```

Augmente de 1 le no des processus qui peuvent entrer

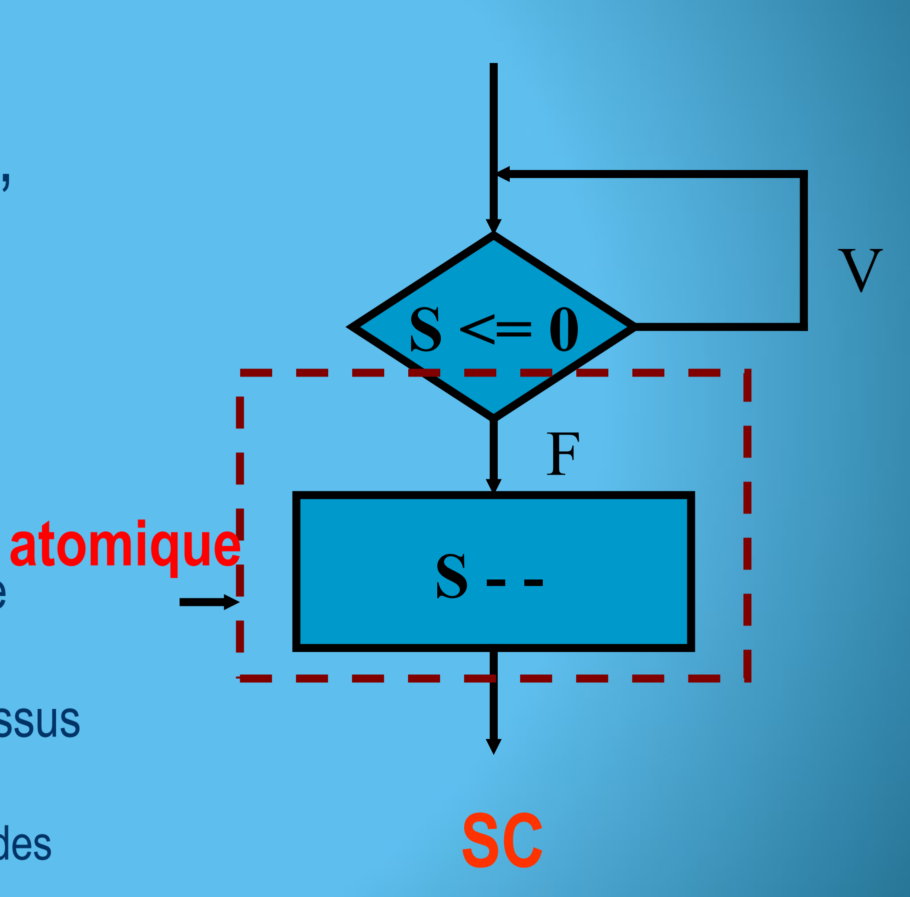
Atomicité

Wait: La séquence test-décrément est atomique, mais pas la boucle!

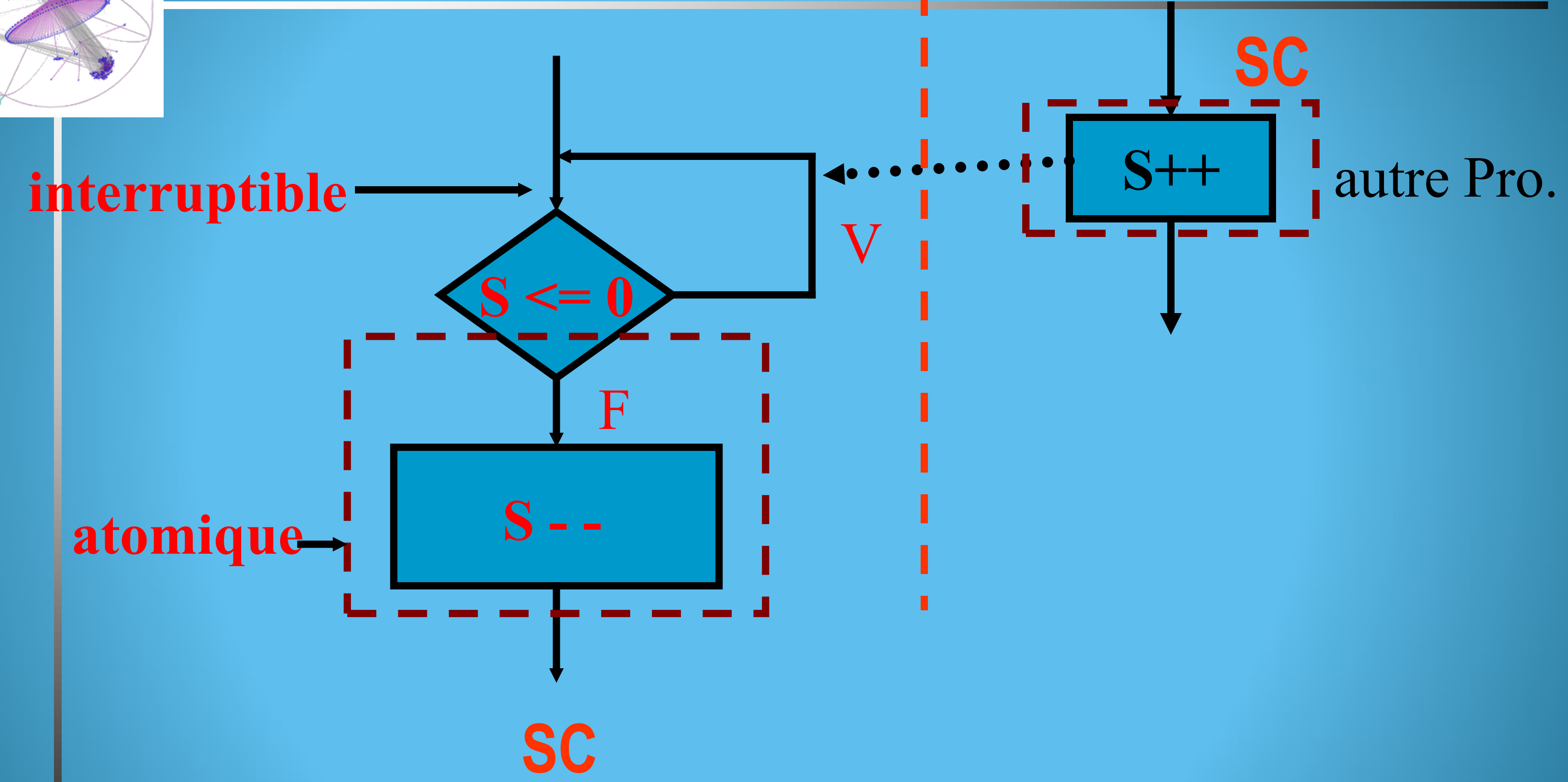
Signal est atomique.

Rappel: les sections atomiques ne peuvent pas être exécutées simultanément par différent processus

(ceci peut être obtenu en utilisant un des mécanismes précédents)



Atomicité et interruptibilité



La boucle n’est pas atomique pour permettre à un autre processus d’interrompre l’attente sortant de la SC

Utilisation des sémaphores pour sections critiques



- Pour n processus
- Initialiser S à 1
- Alors 1 seul processus peut être dans sa SC
- Pour permettre à k processus d’exécuter SC, initialiser S à k

```
processus Ti:
repeat
   wait(S);
    SC
   signal(S);
    SR
forever
```

```
Initialise S à >=1

processus T1:              processus T2:
  repeat                   repeat
  wait(S);                   wait(S);
      SC                       SC
  signal(S);                 signal(S);
      SR                       SR
  forever                  forever
```

Semaphores: vue globale

Peut être facilement généralisé à plus. processus



Utilisation des sémaphores pour synchronisation de processus

- On a 2 processus : T1 et T2
- Énoncé S1 dans T1 doit être exécuté avant énoncé S2 dans T2
- Définissons un sémaphore S
- Initialiser S à 0

- Synchronisation correcte lorsque T1 contient:

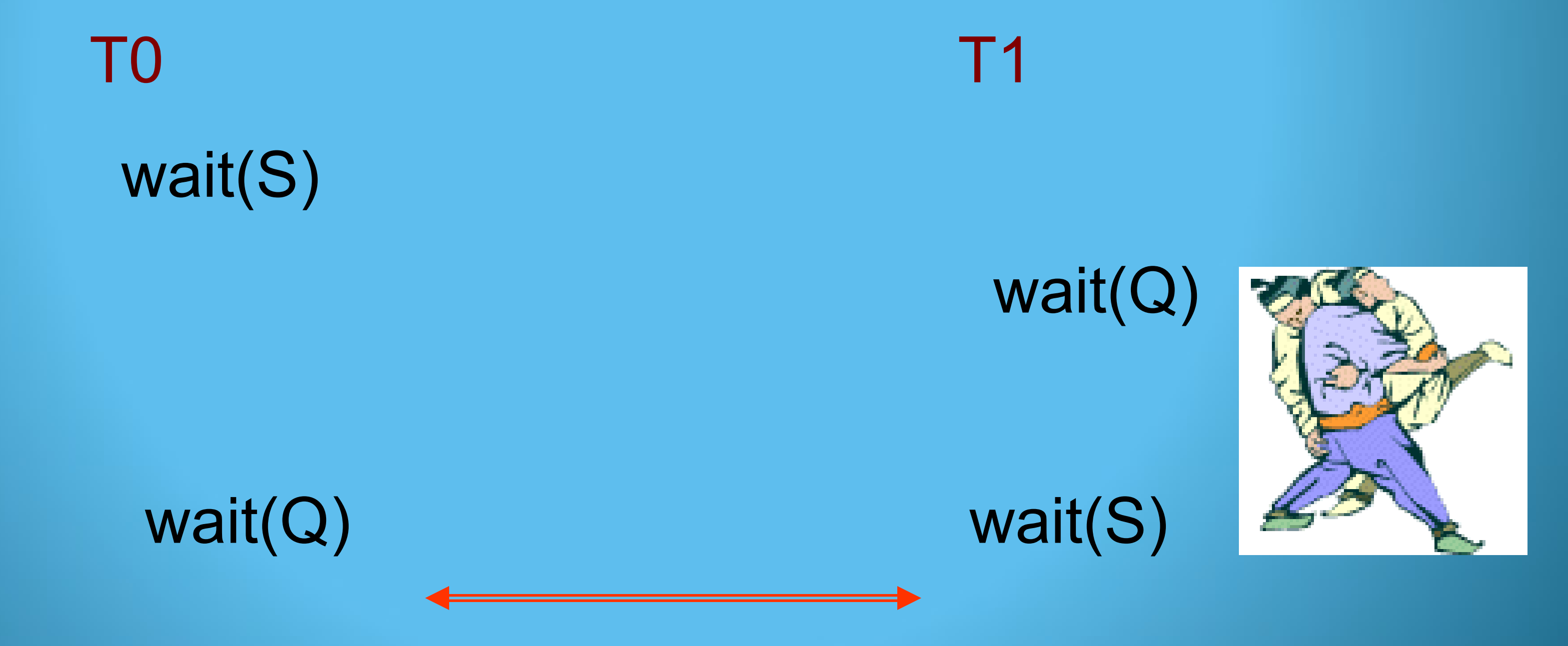
```
S1;
signal(S);
```

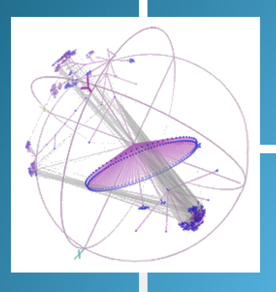
- et que T2 contient:

```
wait(S);
S2;
```

Interblocage et famine avec les sémaphores

- Famine: un processus peut ne jamais arriver à s’exécuter car il ne teste jamais le sémaphore au bon moment
- Interblocage: Supposons S et Q initialisés à 1





Sémaphores: observations

```
wait(S):
while S<=0 {};
 S--;
```

- Quand S >= 0:
  - Le nombre de processus qui peuvent exécuter wait(S) sans devenir bloqués= S
    - S processus peuvent entrer dans la SC
    - noter puissance par rapport à mécanismes déjà vus
    - dans les solutions où S peut être >1il faudra avoir un 2ème sém. pour les faire entrer un à la fois (excl. mutuelle)
- Quand S devient > 1, le processus qui entre le premier dans la SC est le premier à tester S (choix aléatoire)
  - ceci ne sera plus vrai dans la solution suivante
- Quand S < 0: le nombre de processus qui attendent sur S est = |S|

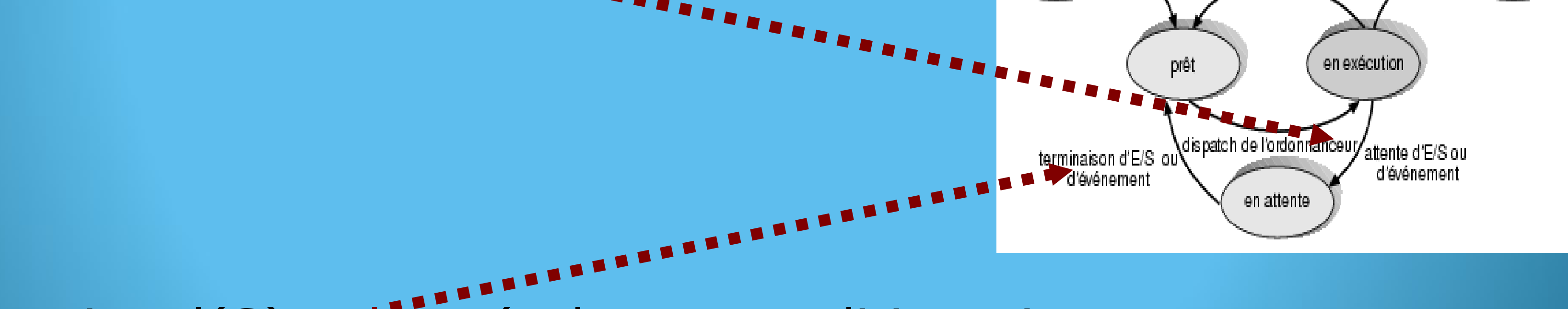


Comment éviter l'attente occupée et le choix aléatoire dans les sémaphores

- Quand un processus doit attendre qu'un sémaphore devienne plus grand que 0, il est mis dans une file d'attente de processus qui attendent sur le même sémaphore.
- Les files peuvent être PAPS (FIFO), avec priorités, etc. Le SE contrôle l`ordre dans lequel les processus entrent dans leur SC.
- *wait* et *signal* sont des appels au SE comme les appels à des opérations d'E/S.
- Il y a une file d'attente pour chaque sémaphore comme il y a une file d'attente pour chaque unité d'E/S.

Sémaphores sans attente occupée

- Un sémaphore S devient une structure de données:
  - Une valeur
  - Une liste d'attente L
- Un processus devant attendre un sémaphore S, est bloqué et ajouté la file d'attente S.L du sémaphore (v. état bloqué = attente chap 4).



- signal(S) enlève (selon une politique juste, ex: PAPS/FIFO) un processus de S.L et le place sur la liste des processus prêts/ready.

Implementation

(les boîtes réprésentent des séquences non-interruptibles)

```
wait(S):   S.value --;
           if S.value < 0    {          // SC occupée
                add this processus to S.L;
                block          // processus mis en état attente (wait)
                }

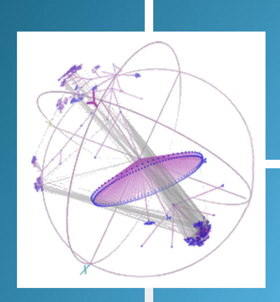
signal(S): S.value ++;
           if S.value ≤ 0 {             // des processus attendent
                remove a process P from S.L;
                wakeup(P)  // processus choisi devient prêt
                }
```

S.value doit être initialisé à une valeur non-négative (dépendant de l'application, v. exemples)



Wait et signal contiennent elles mêmes des SC!

- Les opérations *wait* et *signal* doivent être exécutées atomiquement
- Dans un système avec 1 seule UCT, ceci peut être obtenu en inhibant les interruptions quand un processus exécute ces opérations
- L'attente occupée dans ce cas ne sera pas trop onéreuse car *wait* et *signal* sont brefs



Problèmes classiques de synchronisation

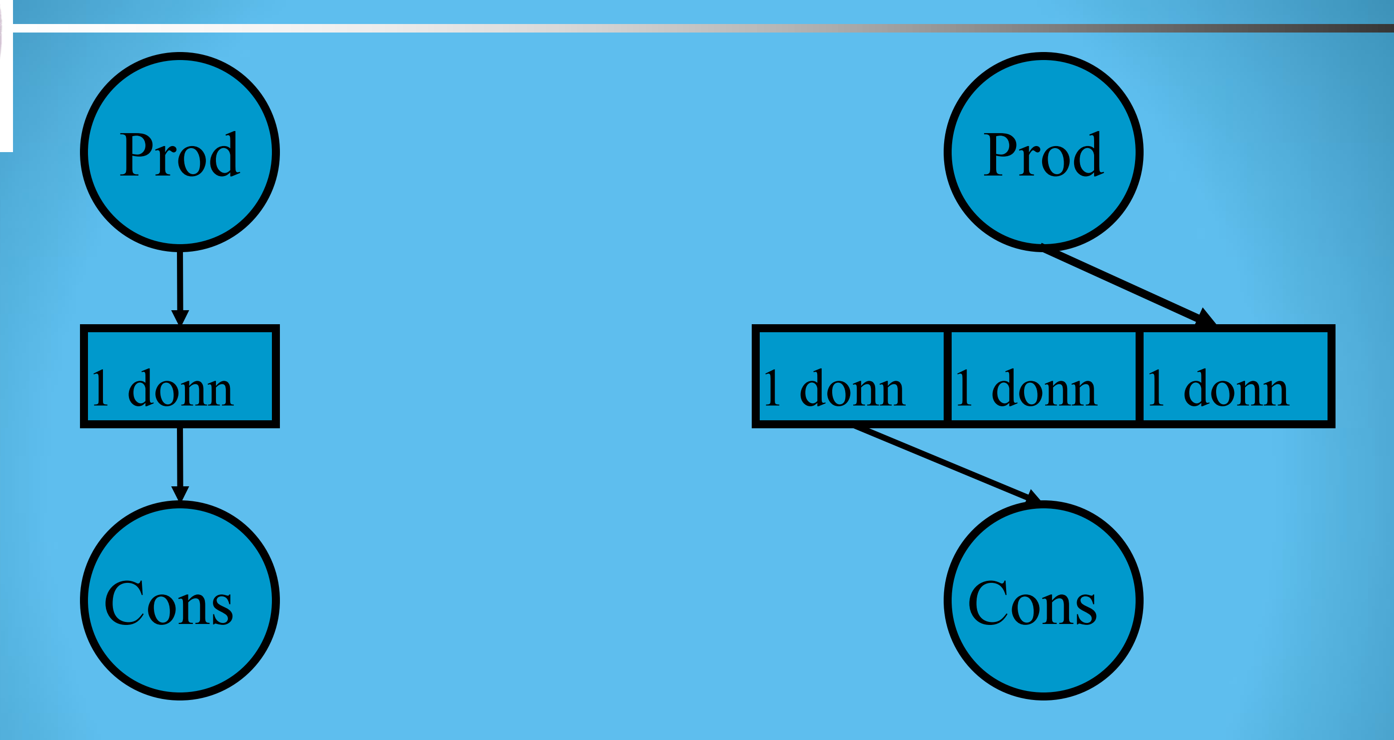
- Tampon borné (producteur-consommateur)
- Écrivains - Lecteurs
- Les philosophes mangeant



Le pb du producteur - consommateur

- **Un problème classique dans l ’étude des processus communicants**
  - un processus *producteur* produit des données (p.ex.des enregistrements d ’un fichier) pour un processus *consommateur*

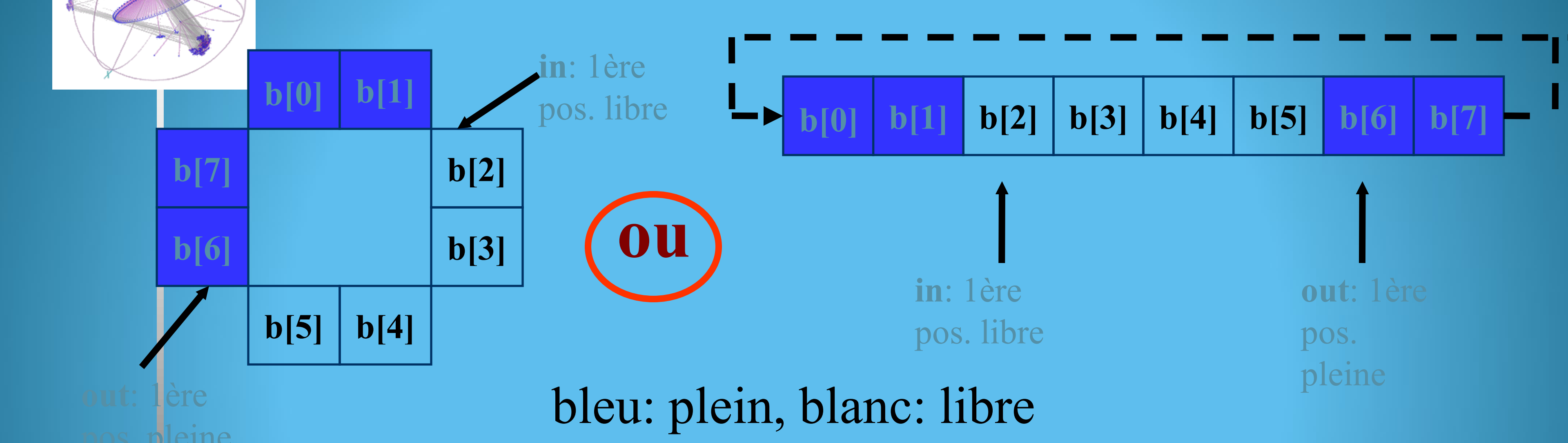
Tampons de communication



Si le tampon est de longueur 1, le producteur et consommateur doivent forcement aller à la même vitesse

Des tampons de longueur plus grandes permettent une certaine indépendance. P.ex. à droite le consommateur a été plus lent

Le tampon borné (bounded buffer)
une structure de données fondamentale dans les SE



Le tampon borné se trouve dans la mémoire partagée entre consommateur et usager



Problème de synchronisation entre processus pour le tampon borné

- **Étant donné que le producteur et le consommateur sont des processus indépendants, des problèmes pourraient se produire en permettant accès simultané au tampon**
- **Les sémaphores peuvent résoudre ce problème**

Sémaphores: rappel.

- **Soit S un sémaphore sur une SC**
  - il est associé à une file d ’attente
  - S positif: S processus peuvent entrer dans SC
  - S zéro: aucun processus ne peut entrer, aucun processus en attente
  - S négatif: |S| processus dans file d ’attente
- **Wait(S): S - -**
  - si après S >= 0, processus peut entrer dans SC
  - si S < 0, processus est mis dans file d ’attente
- **Signal(S): S++**
  - si après S<= 0, il y avait des processus en attente, et un processus est réveillé
- **Indivisibilité = atomicité de ces ops**

Solution avec sémaphores

- Un sémaphore **S** pour **exclusion mutuelle** sur l'accès au tampon
  - Les sémaphores suivants ne font pas l'EM
- Un sémaphore **N** pour synchroniser producteur et consommateur sur le **nombre d'éléments consommables** dans le tampon
- Un sémaphore **E** pour synchroniser producteur et consommateur sur le **nombre d'espaces libres**

Solution de P/C: tampon circulaire fini de dimension k

```
Initialization: S.count=1; //excl. mut.
                N.count=0; //esp. pleins
                E.count=k; //esp. vides
```

```
append(v):
  b[in]=v;
  In ++ mod k;

take():
  w=b[out];
  Out ++ mod k;
  return w;
```

```
Producer:
repeat
  produce v;
  wait(E);
  wait(S);
  append(v);
  signal(S);
  signal(N);
forever
```

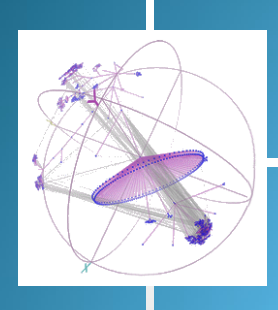
```
Consumer:
repeat
  wait(N);
  wait(S);
  w=take();
  signal(S);
  signal(E);
  consume(w);
forever
```

■ Sections critiques



Points importants à étudier

- **dégâts possibles en interchangeant les instructions sur les sémaphores**
  - ou en changeant leur initialisation
- **Généralisation au cas de plus. Producteurs et consommateur**

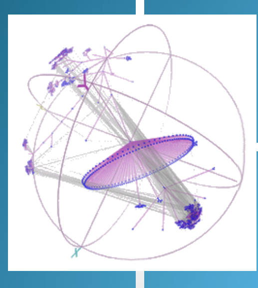


Problème des lecteurs - rédacteurs

- Plusieurs processus peuvent accéder à une base de données
  - Pour y lire ou pour y écrire
- Les rédacteurs doivent être synchronisés entre eux et par rapport aux lecteurs
  - il faut empêcher à un processus de lire pendant l’écriture
  - il faut empêcher à deux rédacteurs d ’écrire simultanément
- Les lecteurs peuvent y accéder simultanément

Une solution (n’exclut pas la famine)

- Variable readcount: nombre de processus lisant la base de données
- Sémaphore mutex: protège la SC où readcount est mis à jour
- Sémaphore wrt: exclusion mutuelle entre rédacteurs et lecteurs
- Les rédacteurs doivent attendre sur wrt
  - les uns pour les autres
  - et aussi la fin de toutes les lectures
- Les lecteurs doivent
  - attendre sur wrt quand il y a des rédacteurs qui écrivent
  - bloquer les rédacteurs sur wrt quand il y a des lecteurs qui lisent
  - redémarrer les rédacteurs quand personne ne lit



Les données et les rédacteurs

Données: deux sémaphores et une variable

```
mutex, wrt: semaphore (init. 1);
readcount : integer (init. 0);
```

Rédacteur

```
wait(wrt);
            . . .
      // écriture
            . . .
signal(wrt);
```

Les lecteurs

```
wait(mutex);
        readcount ++ ;
        if readcount == 1 then wait(wrt);
signal(mutex);
```

Le premier lecteur d 'un groupe pourrait devoir attendre sur wrt, il doit aussi bloquer les rédacteurs. Quand il sera entré, les suivants pourront entrer librement

```
//SC: lecture
```

```
wait(mutex);
        readcount -- ;
        if readcount == 0 then signal(wrt);
signal(mutex):
```

Le dernier lecteur sortant doit permettre l`accès aux rédacteurs

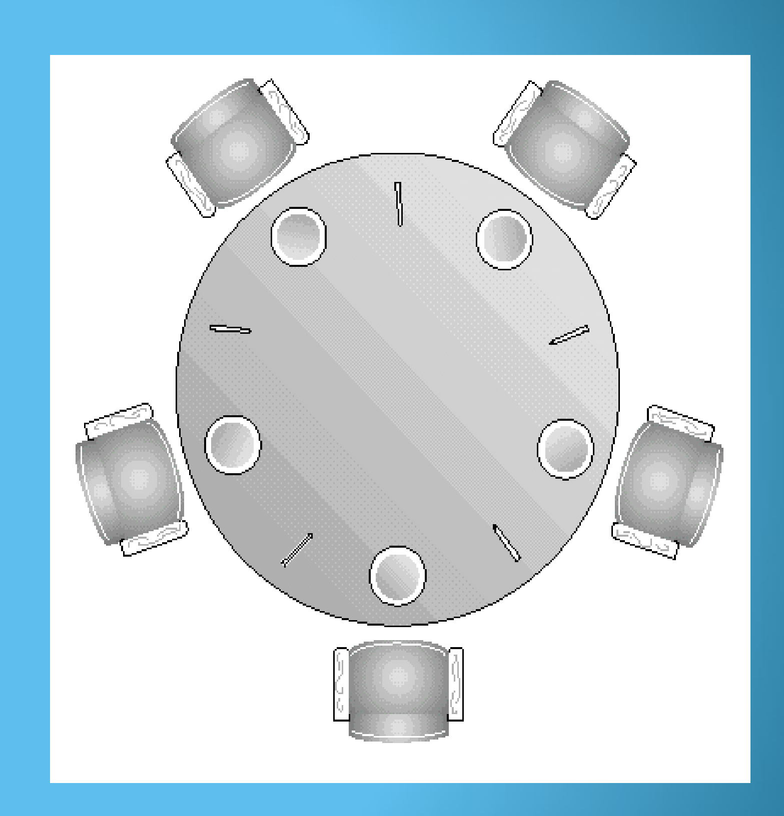


Observations

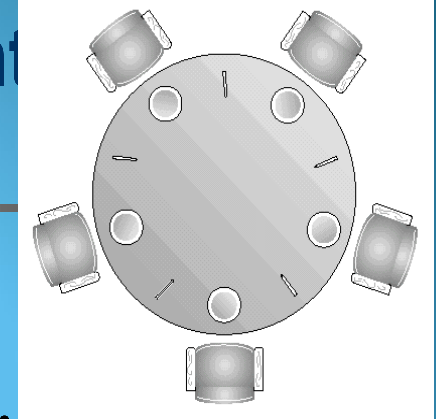
- Le 1er lecteur qui entre dans la SC bloque les rédacteurs (wait (wrt)), le dernier les remet en marche (signal (wrt))
- Si 1 rédacteur est dans la SC, 1 lecteur attend sur wrt, les autres sur mutex
- un signal(wrt) peut faire exécuter un lecteur ou un rédacteur

Le problème des philosophes mangeant

- 5 philosophes qui mangent et pensent
- Pour manger il faut 2 fourchettes, droite et gauche
- On en a seulement 5!
- Un problème classique de synchronisation
- Illustre la difficulté d'allouer ressources aux processus tout en évitant interblocage et famine



Le problème des philosophes mangeant



- Un processus par philosophe
- Un sémaphore par fourchette:
  - fork: array[0..4] of semaphores
  - Initialisation: fork[i] =1 for i:=0..4
- Première tentative:
  - interblocage si chacun débute en prenant sa fourchette gauche!
    - `Wait(fork[i])`

```
processus Pi:
repeat
  think;
  wait(fork[i]);
  wait(fork[i+1 mod 5]);
  eat;
  signal(fork[i+1 mod 5]);
  signal(fork[i]);
forever
```

Le problème des philosophes mangeant

- Une solution: admettre seulement 4 philosophes à la fois qui peuvent tenter de manger
- Il y aura toujours au moins 1 philosophe qui pourra manger
  - même si tous prennent 1 fourchette
- Ajout d'un sémaphore T qui limite à 4 le nombre de philosophes "assis à la table"
  - initial. de T à 4
- N'empêche pas famine!

```
processus Pi:
repeat
  think;
  wait(T);
  wait(fork[i]);
  wait(fork[i+1 mod 5]);
  eat;
  signal(fork[i+1 mod 5]);
  signal(fork[i]);
  signal(T);
forever
```



Avantage des sémaphores
(par rapport aux solutions précédentes)

- Une seule variable partagée par section critique
- deux seules opérations: wait, signal
- contrôle plus localisé (que avec les précédents)
- extension facile au cas de plus. processus
- possibilité de faire entrer plus. processus à la fois dans une section critique
- gestion de files d`attente par le SE: famine évitée si le SE est équitable (p.ex. files FIFO)

Problème avec sémaphores:
difficulté de programmation

- wait et signal sont dispersés parmi plusieurs processus, mais ils doivent se correspondre
  - V. programme du tampon borné
- Utilisation doit être correcte dans tous les processus
- Un seul “mauvais” processus peut faire échouer toute une collection de processus (p.ex. oublie de faire signal)
- Considérez le cas d`un processus qui a des waits et signals dans des boucles et des tests...



Le problème de la SC en pratique...

- Les systèmes réels rendent disponibles plusieurs mécanismes qui peuvent être utilisés pour obtenir la solution la plus efficace dans différentes situations